BRUCE Vining SERVICES L.L.C. Integrated solutions for the System i user community

CL – The Story Continues

Gateway/400 Group: December 2009
Bruce Vining
bvining@brucevining.com

In This Session ...

BRUCE Vining SERVICES L.L.C. Integrated solutions for the System i user community

- We will review some of the significant enhancements that have been made in CL during the last three releases.
- By the end of this session, attendees will be able to:
  - Use new data types such as integer and pointer
  - Use multiple files in one program
  - Use programming constructs such as:
    - DoFor, DoWhile, DoUntil
    - Subroutines
  - Use structures and based variables
  - Use new compiler options

2

What We'll Cover …

- Integers
- Use Multiple Files in One Program
- Programming Constructs
- Pointers and Based Variables
- Structures
- Compiler Options
- Wrap-up

3

Integers

- Direct support for signed and unsigned variables with V5R3
  - `Dcl        Var(&Signed)    Type(*Int)`
  - `Dcl        Var(&Unsigned)  Type(*UInt)`
- Much nicer than using the previous %Bin built-in support
  - The "old" way:

```
Dcl        Var(&Char)          Type(*Char)  Len(4)
ChgVar     Var(%Bin(&Char))    Value(10)
```

  - The "new" way:

```
Dcl        Var(&Signed)        Type(*Int)
ChgVar     Var(&Signed)        Value(10)
```

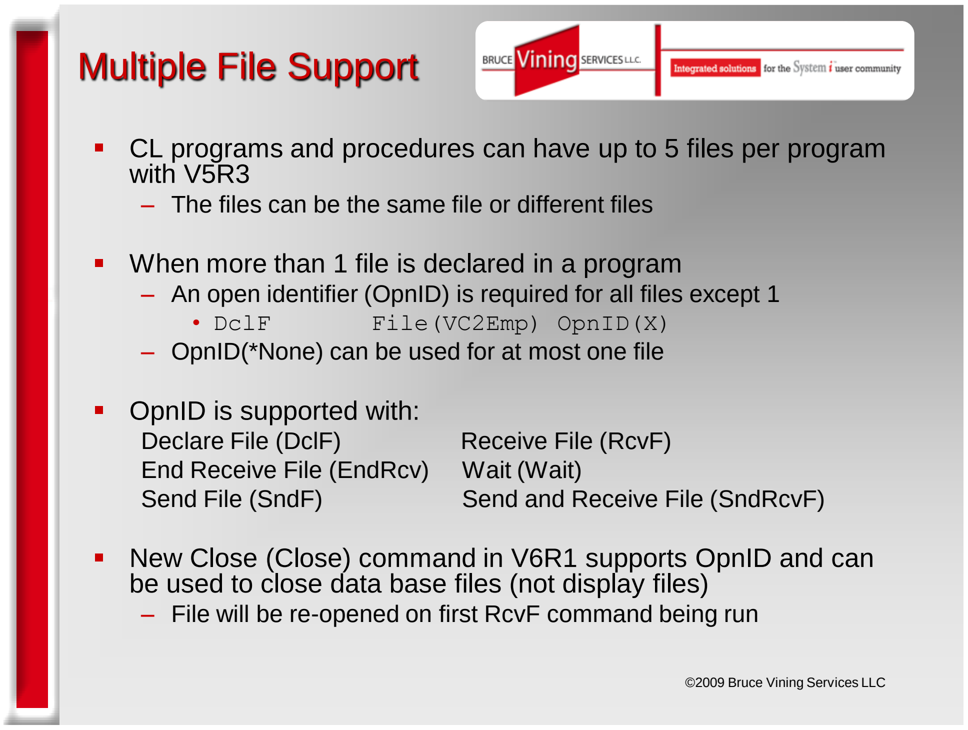
- Much more productive debug assistance also
  - `Eval &Char` displays `"blobs"`
  - `Eval &Char:x` displays `0000000A`
  - `Eval &Signed` displays `10`

Integers (continued)

- Integers can be 2 or 4-bytes in length
  - With 2-byte integers (Len(2))
    - Signed values from -32,768 to 32,767
    - Unsigned values can be from 0 to 65,535
  - With 4-byte integers (Len(4))
    - Signed values from -2,147,483,648 to 2,147,483,647
    - Unsigned values from 0 to 4,294,967,295
  - The default is a length of 4 bytes
- V5R4 integer support on DclF command and Declare Binary Fields (DclBinFld) keyword
  - `DclF  File(VC2Emp) DclBinFld(*Int)`
  - For compatibility DclBinFld defaults to *Dec

What We'll Cover …

- Integers
- Use Multiple Files in One Program
- Programming Constructs
- Pointers and Based Variables
- Structures
- Compiler Options
- Wrap-up

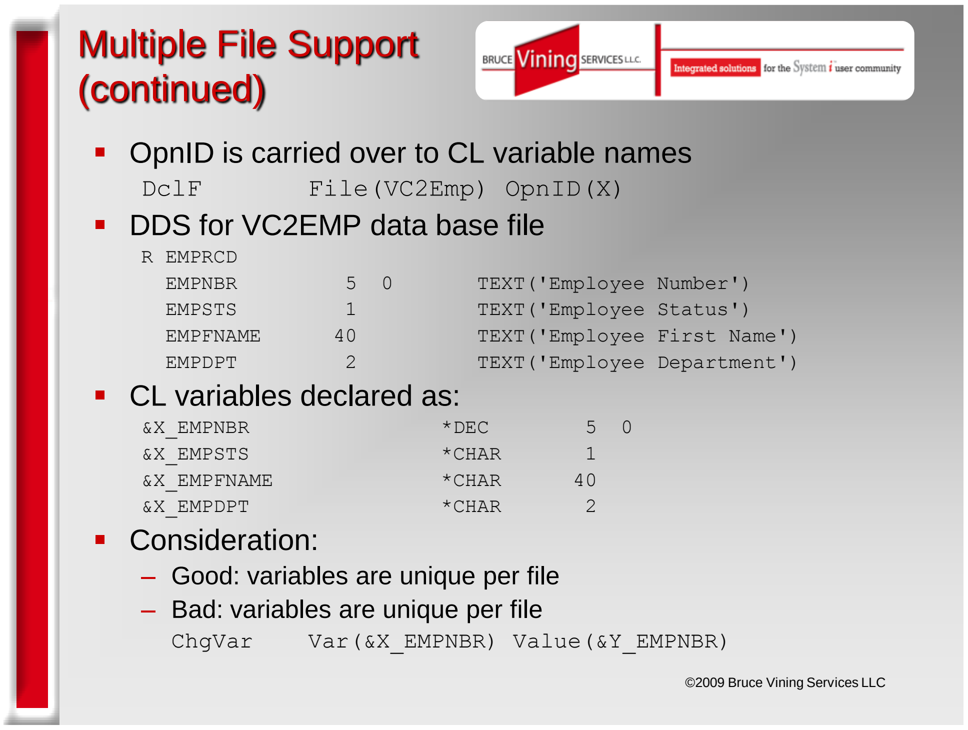


Multiple File Support

- CL programs and procedures can have up to 5 files per program with V5R3
  - The files can be the same file or different files
- When more than 1 file is declared in a program
  - An open identifier (OpnID) is required for all files except 1
    - `DclF        File(VC2Emp)  OpnID(X)`
  - OpnID(*None) can be used for at most one file
- OpnID is supported with:

Declare File (DclF)	Receive File (RcvF)
End Receive File (EndRcv)	Wait (Wait)
Send File (SndF)	Send and Receive File (SndRcvF)

- New Close (Close) command in V6R1 supports OpnID and can be used to close data base files (not display files)
  - File will be re-opened on first RcvF command being run



Multiple File Support (continued)

- OpnID is carried over to CL variable names

```
DclF        File(VC2Emp)  OpnID(X)
```

- DDS for VC2EMP data base file

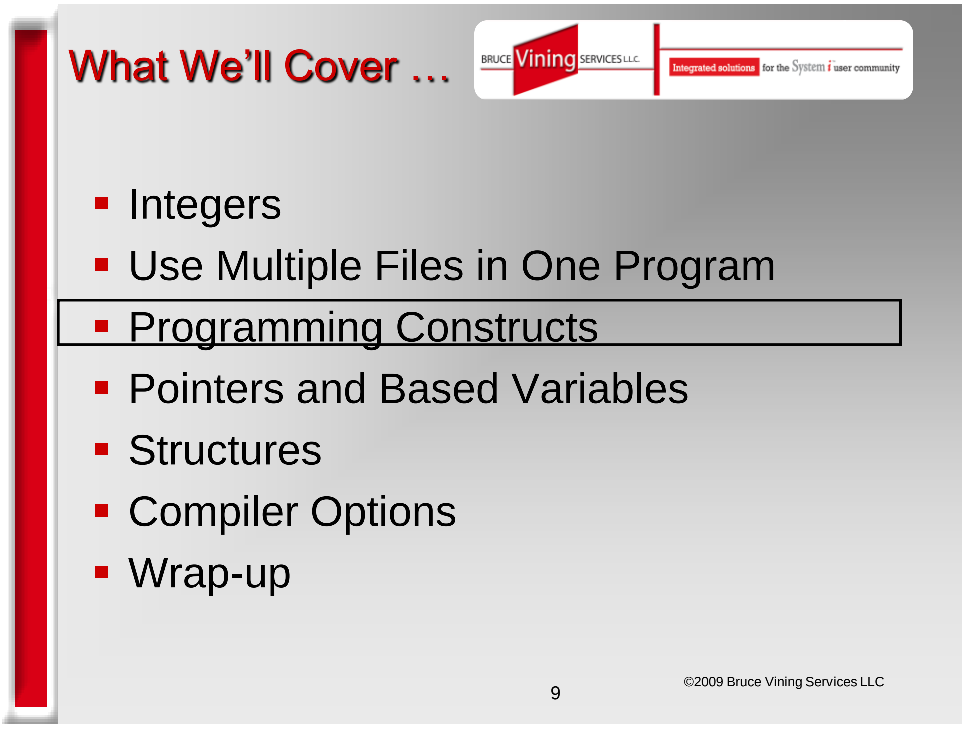
```
R EMPRCD
  EMPNBR          5  0        TEXT('Employee Number')
  EMPSTS          1           TEXT('Employee Status')
  EMPFNAME       40           TEXT('Employee First Name')
  EMPDPT          2           TEXT('Employee Department')
```

- CL variables declared as:

```
&X_EMPNBR                 *DEC         5  0
&X_EMPSTS                 *CHAR        1
&X_EMPFNAME               *CHAR       40
&X_EMPDPT                 *CHAR        2
```

- Consideration:
  - Good: variables are unique per file
  - Bad: variables are unique per file

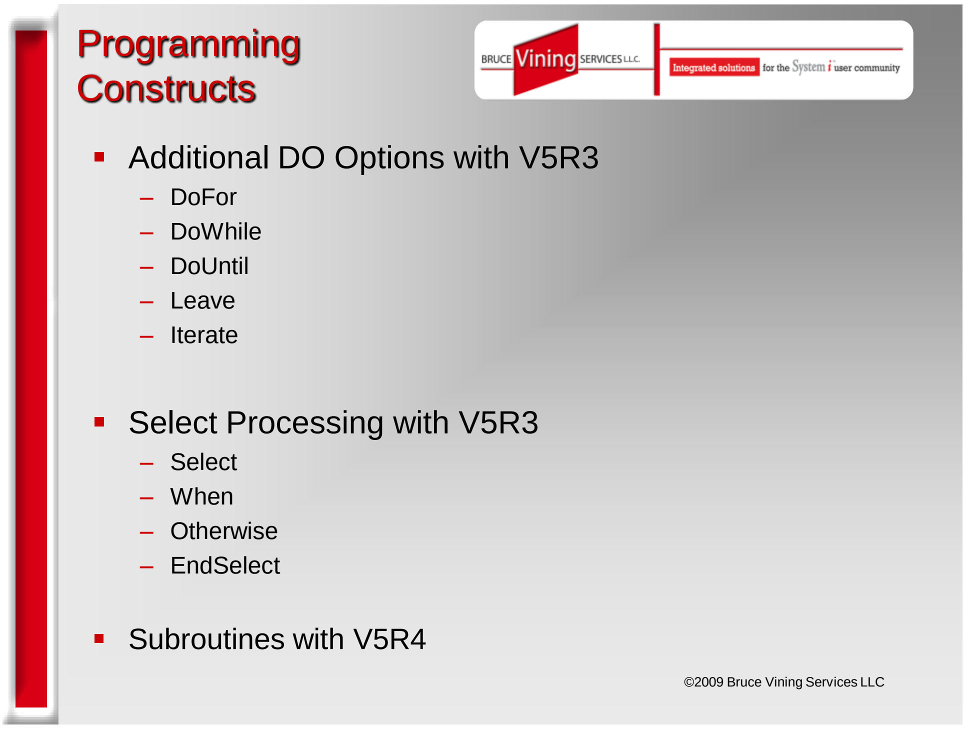
```
ChgVar     Var(&X_EMPNBR)  Value(&Y_EMPNBR)
```



What We'll Cover …

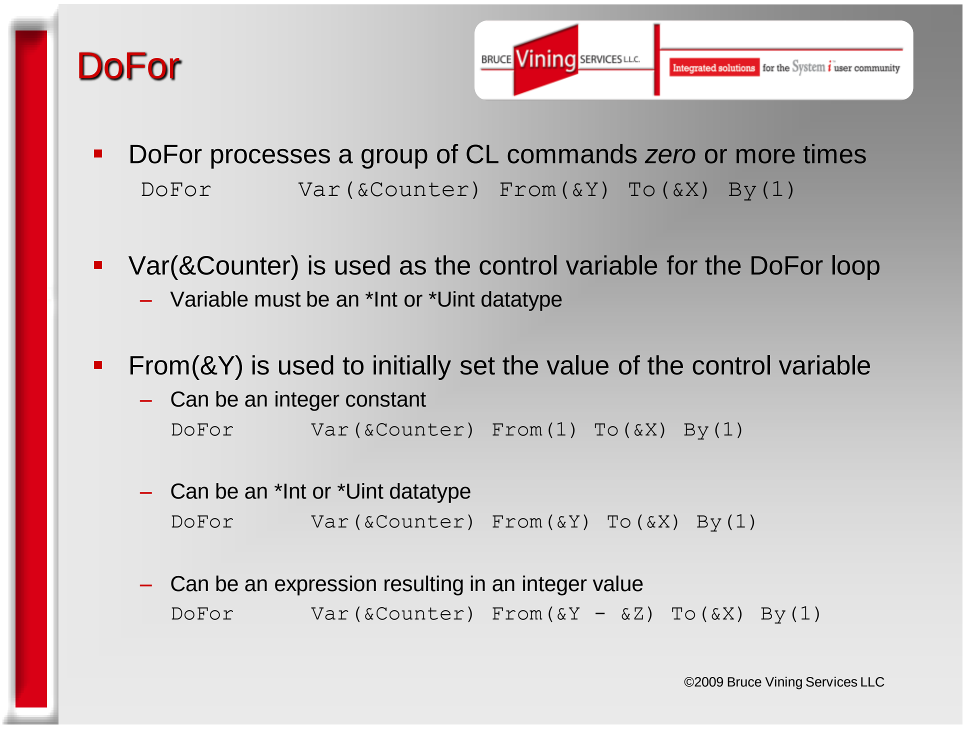
- Integers
- Use Multiple Files in One Program
- Programming Constructs
- Pointers and Based Variables
- Structures
- Compiler Options
- Wrap-up

9



Programming Constructs

- Additional DO Options with V5R3
  - DoFor
  - DoWhile
  - DoUntil
  - Leave
  - Iterate
- Select Processing with V5R3
  - Select
  - When
  - Otherwise
  - EndSelect
- Subroutines with V5R4



DoFor

- DoFor processes a group of CL commands *zero* or more times

```
 DoFor       Var(&Counter)  From(&Y)  To(&X)  By(1)
```

- Var(&Counter) is used as the control variable for the DoFor loop
  - Variable must be an *Int or *Uint datatype
- From(&Y) is used to initially set the value of the control variable
  - Can be an integer constant

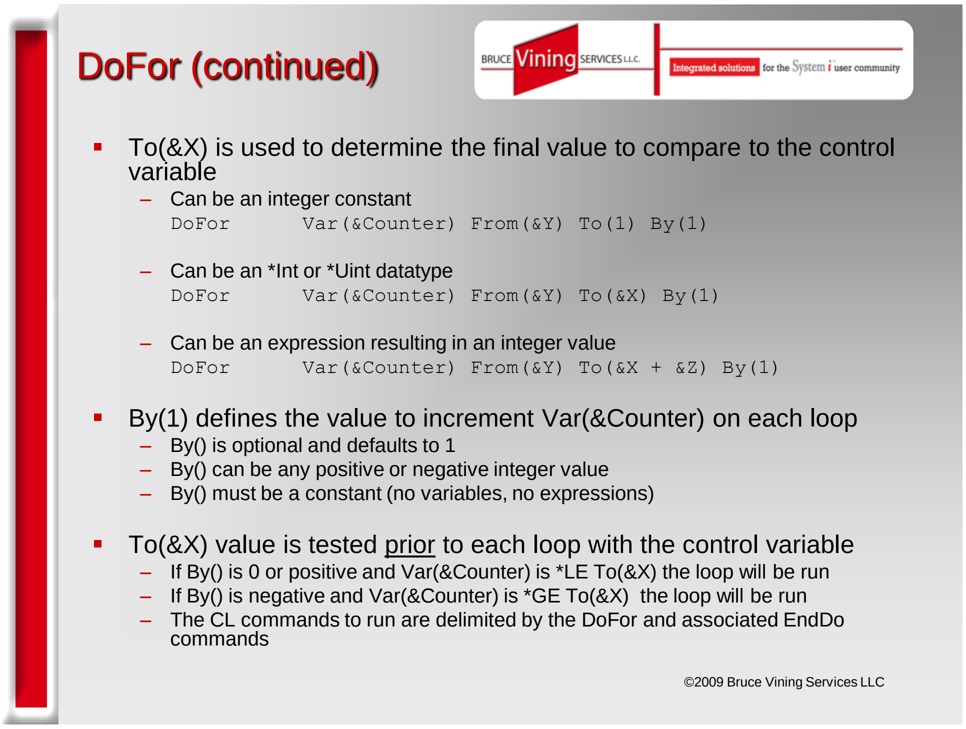
```
    DoFor       Var(&Counter)  From(1)  To(&X)  By(1)
```

  - Can be an *Int or *Uint datatype

```
    DoFor       Var(&Counter)  From(&Y)  To(&X)  By(1)
```

  - Can be an expression resulting in an integer value

```
    DoFor       Var(&Counter)  From(&Y - &Z)  To(&X)  By(1)
```



DoFor (continued)

- To(&X) is used to determine the final value to compare to the control variable
  - Can be an integer constant

```
    DoFor       Var(&Counter)  From(&Y)  To(1)  By(1)
```

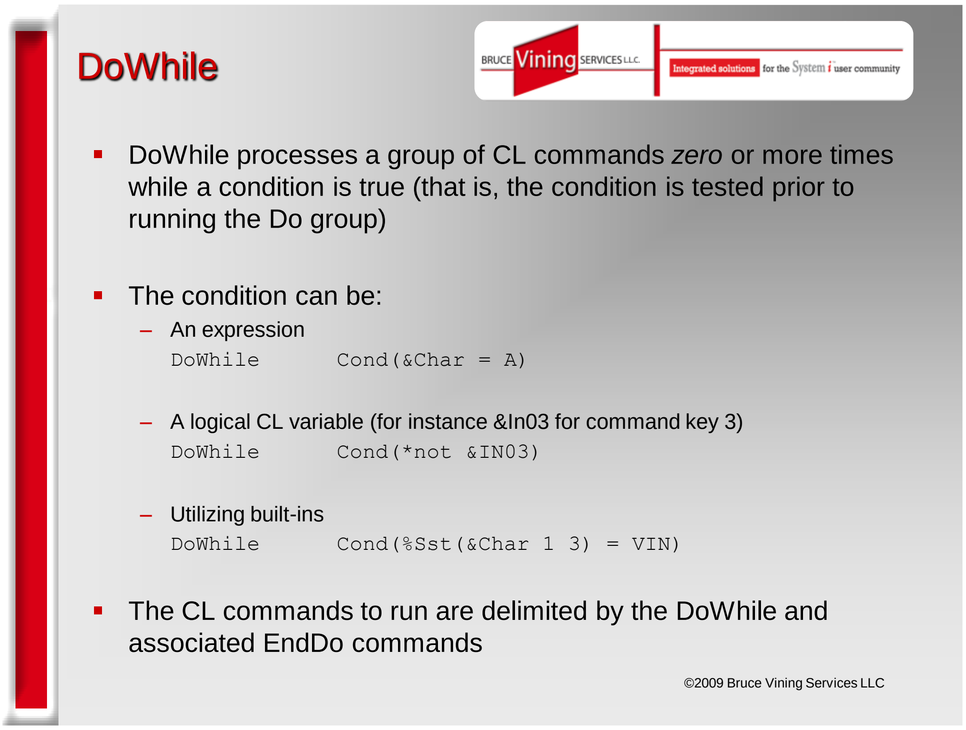
  - Can be an *Int or *Uint datatype

```
    DoFor       Var(&Counter)  From(&Y)  To(&X)  By(1)
```

  - Can be an expression resulting in an integer value

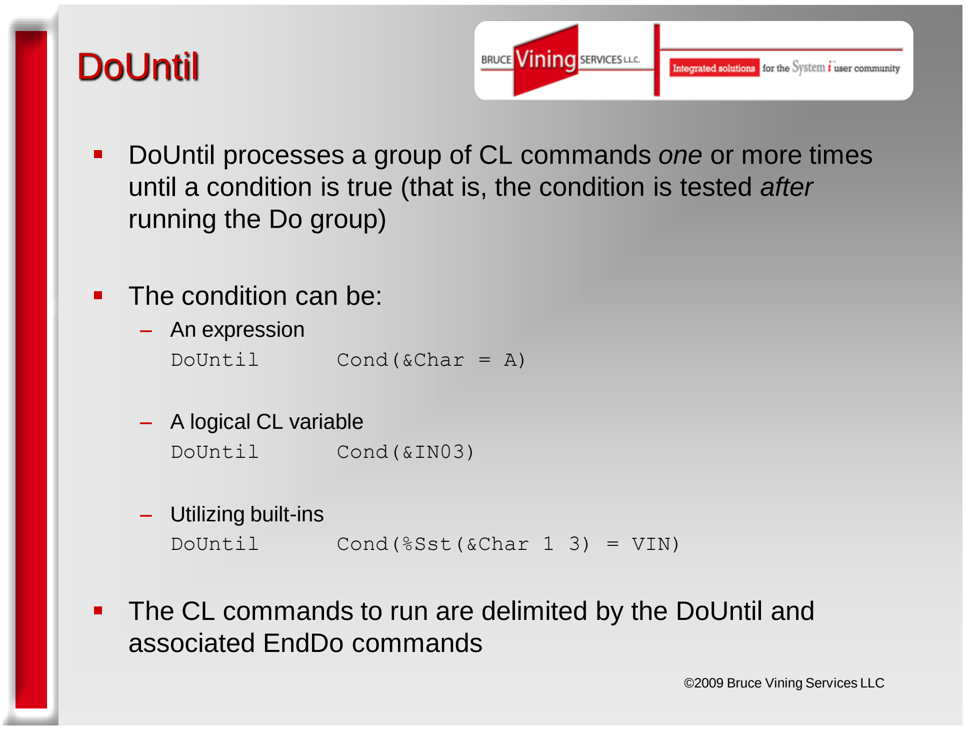
```
    DoFor       Var(&Counter)  From(&Y)  To(&X + &Z)  By(1)
```

- By(1) defines the value to increment Var(&Counter) on each loop
  - By() is optional and defaults to 1
  - By() can be any positive or negative integer value
  - By() must be a constant (no variables, no expressions)
- To(&X) value is tested prior to each loop with the control variable
  - If By() is 0 or positive and Var(&Counter) is *LE To(&X) the loop will be run
  - If By() is negative and Var(&Counter) is *GE To(&X) the loop will be run
  - The CL commands to run are delimited by the DoFor and associated EndDo commands



DoWhile

- DoWhile processes a group of CL commands *zero* or more times while a condition is true (that is, the condition is tested prior to running the Do group)
- The condition can be:
  - An expression
    ```
    DoWhile      Cond(&Char = A)
    ```
  - A logical CL variable (for instance &In03 for command key 3)
    ```
    DoWhile      Cond(*not &IN03)
    ```
  - Utilizing built-ins
    ```
    DoWhile      Cond(%Sst(&Char 1 3) = VIN)
    ```
- The CL commands to run are delimited by the DoWhile and associated EndDo commands



DoUntil

- DoUntil processes a group of CL commands *one* or more times until a condition is true (that is, the condition is tested *after* running the Do group)
- The condition can be:
  - An expression
    ```
    DoUntil      Cond(&Char = A)
    ```
  - A logical CL variable
    ```
    DoUntil      Cond(&IN03)
    ```
  - Utilizing built-ins
    ```
    DoUntil      Cond(%Sst(&Char 1 3) = VIN)
    ```
- The CL commands to run are delimited by the DoUntil and associated EndDo commands

Leave

- Leave *exits* the active DoFor, DoWhile, or DoUntil CL command group(s).
  - Leave CmdLbl(*CURRENT) causes the program to exit the current DoXXXXX group of CL commands and run the CL command following the associated EndDo command. This is the default.
  - Leave CmdLbl(XYZ) causes the program to exit all active DoXXXXX groups of CL commands at or imbedded within the DoXXXXX group labeled XYZ

```
XYZ: DoWhile    Cond(&Char = A)
                DoFor Var(&Counter) From(&Y) To(&X)
                      If Cond(&In03) Then(Leave CmdLbl(XYZ))

                      /* Do some work */

                      If Cond(&Status = ABCxxx) Then(Leave)

                      /* Do more work */

                      EndDo

                /* Do even more work... */

                EndDo
     /* And some more... */
```

Iterate

- Iterate immediately passes control to the associated EndDo command and retests the condition of the associated DoFor, DoWhile, or DoUntil group
  - Iterate CmdLbl(*CURRENT) causes the program to retest the current DoXXXXX condition. This is the default.
  - Iterate CmdLbl(XYZ) causes the program to retest the DoXXXXX condition labeled XYZ

```
XYZ: DoWhile    Cond(&Char = A)
                DoFor Var(&Counter) From(&Y) To(&X)
                      If Cond(&In03) Then(Iterate CmdLbl(XYZ))

                      /* Do some work */

                      If Cond(&Status = ReTest) Then(Iterate)
                      If Cond(&Status = GetOut) Then(Leave)

                      /* Do more work */

                      EndDo

                /* Do even more work... */

                EndDo
     /* And some more... */
```

Select Groups

- The Select command begins a control structure for conditional processing
- The When command identifies a condition to be tested
  - One or more When commands can be defined in a Select group
  - When commands are tested in the order found in the Select group
  - When commands are *mutually exclusive*. If one When condition tests true then no additional When conditions are tested. So the ordering of the When conditions can be very important
  - Processing resumes after the associated EndSelect command
- The OtherWise command identifies the CL command to be run if no When condition tests true
  - OtherWise is not required in a Select group
  - I highly recommend having one though
- The EndSelect command defines the end of the current Select group
- Select groups can be nested

Select Groups (continued)

```
DoWhile    Cond(&More_Input)
           Select
              When Cond(&In03) Then(Return)
              When Cond(&In12) Then(Leave)
              When Cond(&Action = Yes) Then(Do)
                   /* Do appropriate work */
                   EndDo
              When Cond(&Action = Maybe) Then(Do)
                   /* Do appropriate work */
                   EndDo
              When Cond((&Action = No) *And +
                     (&Stat *LT 10)) Then(Do)
                   /* Do appropriate work */
                   EndDo
              When Cond((&Action = No) *And +
                     (&Stat *GE 10)) Then(Do)
                   /* Do appropriate work */
                   EndDo
              OtherWise  Cmd(Do)
                   /* Do appropriate work */
                   EndDo
              EndSelect
           /* Do appropriate work after all conditions handled */
           EndDo
```

Select Groups (continued)

- Select groups are my personal favorite (of this section of the session that is)
- Avoids nesting If/Else logic

```
DoWhile     Cond(&More_Input)
            If Cond(&In03) Then(Return)
            If Cond(&In12) Then(Leave)
            If Cond(&Action = Yes) Then(Do)
               /* Do appropriate work */
               EndDo
            Else Cmd(If Cond(&Action = Maybe) Then(Do))
                     /* Do appropriate work */
                     EndDo
                 If Cond((&Action = No) *And +
                         (&Stat *LT 10)) Then(Do)
                    /* Do appropriate work */
                    EndDo
                 Else Cmd(…… Just more of the same……)
            /* Do appropriate work after all the 'If's  */
            EndDo
```

  - Easier to read and follow (for me anyway)

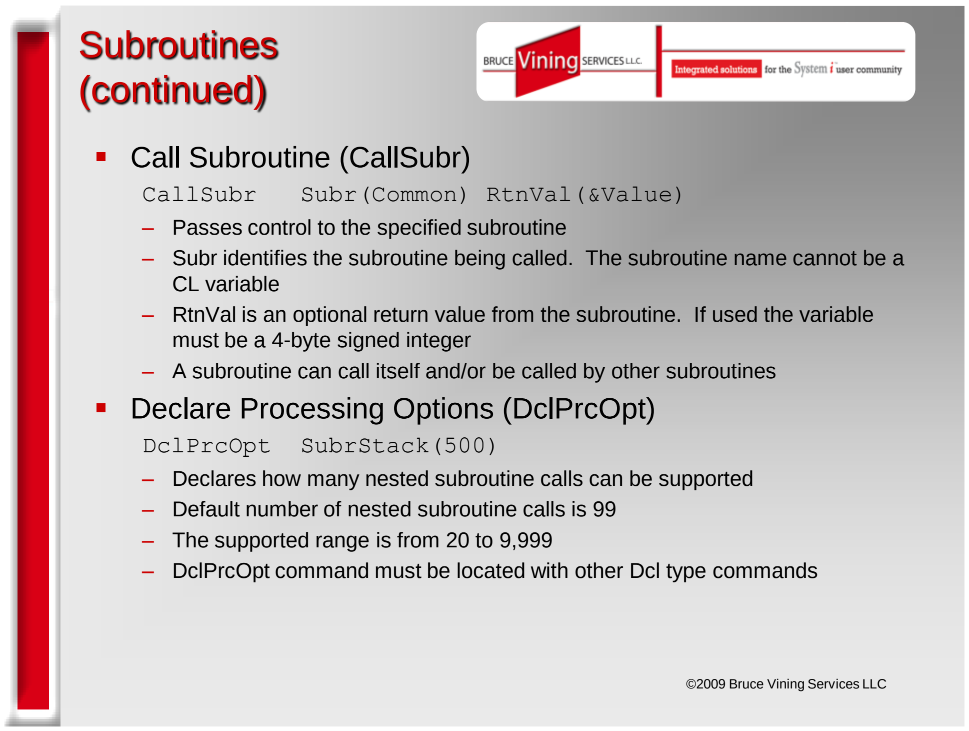
- Avoids many GoTo commands to a common end of the If logic if trying to avoid nested If/Else logic
- OtherWise makes sure I consider “what if”
- Easy to start utilizing

Subroutines

- Subroutines provide for sharing of CL program code within a procedure
- Subroutines cannot declare local variables
- Subroutines cannot be passed parameter values
- Subroutines can return a value to the caller of the subroutine
- Subroutines are physically found in the CL source program after the main line commands and before the EndPgm command

```
Pgm
Dcls, DclFs, CopyRight, etc
. . .
CallSubr   Subr(Common)
. . .
Return
Subr       Subr(Common)
. . .
EndSubr

EndPgm
```



Subroutines (continued)

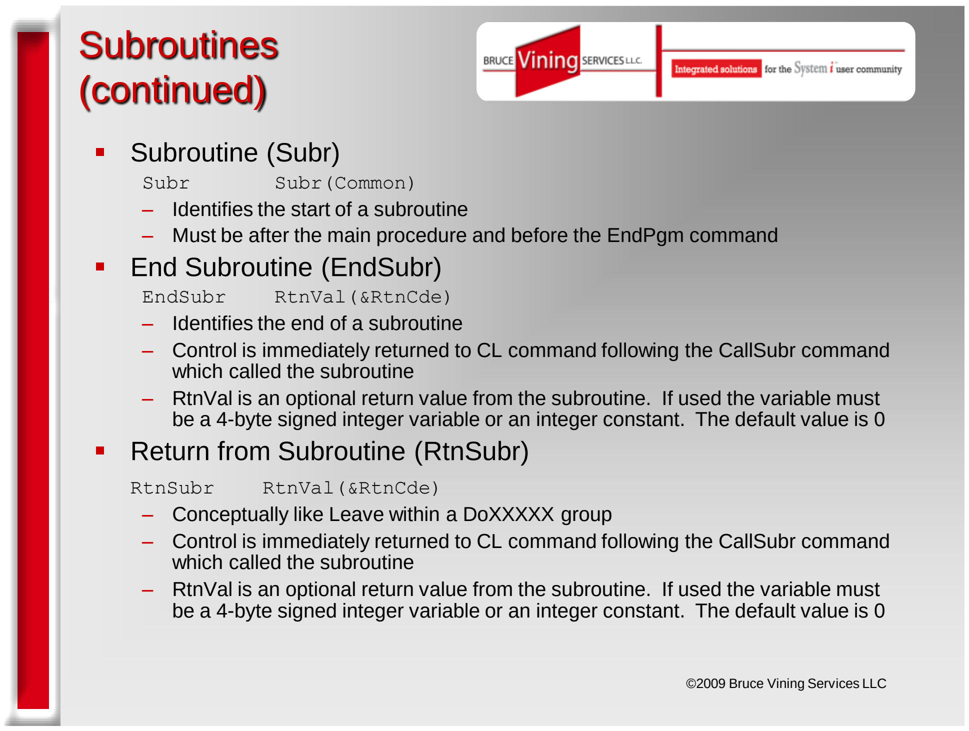
- Call Subroutine (CallSubr)

```
CallSubr   Subr(Common)  RtnVal(&Value)
```

  - Passes control to the specified subroutine
  - Subr identifies the subroutine being called. The subroutine name cannot be a CL variable
  - RtnVal is an optional return value from the subroutine. If used the variable must be a 4-byte signed integer
  - A subroutine can call itself and/or be called by other subroutines
- Declare Processing Options (DclPrcOpt)

```
DclPrcOpt  SubrStack(500)
```

  - Declares how many nested subroutine calls can be supported
  - Default number of nested subroutine calls is 99
  - The supported range is from 20 to 9,999
  - DclPrcOpt command must be located with other Dcl type commands



Subroutines (continued)

- Subroutine (Subr)

```
Subr        Subr(Common)
```

  - Identifies the start of a subroutine
  - Must be after the main procedure and before the EndPgm command
- End Subroutine (EndSubr)

```
EndSubr     RtnVal(&RtnCde)
```

  - Identifies the end of a subroutine
  - Control is immediately returned to CL command following the CallSubr command which called the subroutine
  - RtnVal is an optional return value from the subroutine. If used the variable must be a 4-byte signed integer variable or an integer constant. The default value is 0
- Return from Subroutine (RtnSubr)

```
RtnSubr     RtnVal(&RtnCde)
```

  - Conceptually like Leave within a DoXXXXX group
  - Control is immediately returned to CL command following the CallSubr command which called the subroutine
  - RtnVal is an optional return value from the subroutine. If used the variable must be a 4-byte signed integer variable or an integer constant. The default value is 0

What We'll Cover …

BRUCE Vining SERVICES LLC | Integrated solutions for the System i user community

- Integers
- Use Multiple Files in One Program
- Programming Constructs
- Pointers and Based Variables
- Structures
- Compiler Options
- Wrap-up

Pointers – Some Background

BRUCE Vining SERVICES LLC | Integrated solutions for the System i user community

- Assume you have these DCLs in a program:

```
Dcl Var(&Text) Type(*Char) Len(20) Value('Some text')
Dcl Var(&More) Type(*Char) Len(5)  Value('ABC')
Dcl Var(&OK)   Type(*Lgl)          Value('1')
Dcl Var(&Code) Type(*Char) Len(1)  Value('X')
```

- Then in memory (activation group) there is *conceptually*:

```
????????Some~text~~~~~~~~~~~ABC~~1X????????????????
```

  – With ? representing a variable value for another program that is active in your job
  – And ~ is a blank within your program variable

- A pointer is a variable that is set to the address of your variable within memory
  – If the address of the first ? is decimal 12345678 then the address of &Text is decimal 12345686 (12345678 + 8) as there are 8 ?s.

Pointers – Background (continued)

BRUCE Vining SERVICES LLC | Integrated solutions for the System i user community

- If you have ever called a program or run a command you have used pointers without (necessarily) knowing it

```
Dcl  Var(&Text) Type(*Char) Len(20) Value('Some text')
Call Pgm(ABC) Parm(&Text)
```

- Under the covers the Call command is passing a pointer to the &Text variable (ie, the address of &Text)

```
Pgm  Parm(&Text)
Dcl  Var(&Text) Type(*Char) Len(20)
```

- Which is why:
  - Variable names do not have to be the same across programs
  - Variable definitions do not have to match (though they should)
  - Changes made to &Text by program ABC are immediately reflected in the calling program (as &Text really is in the calling programs memory)

Pointer Variables

BRUCE Vining SERVICES LLC | Integrated solutions for the System i user community

- Direct support for pointer variables with V5R4

```
Dcl    Var(&MyPointer)    Type(*Ptr)
```

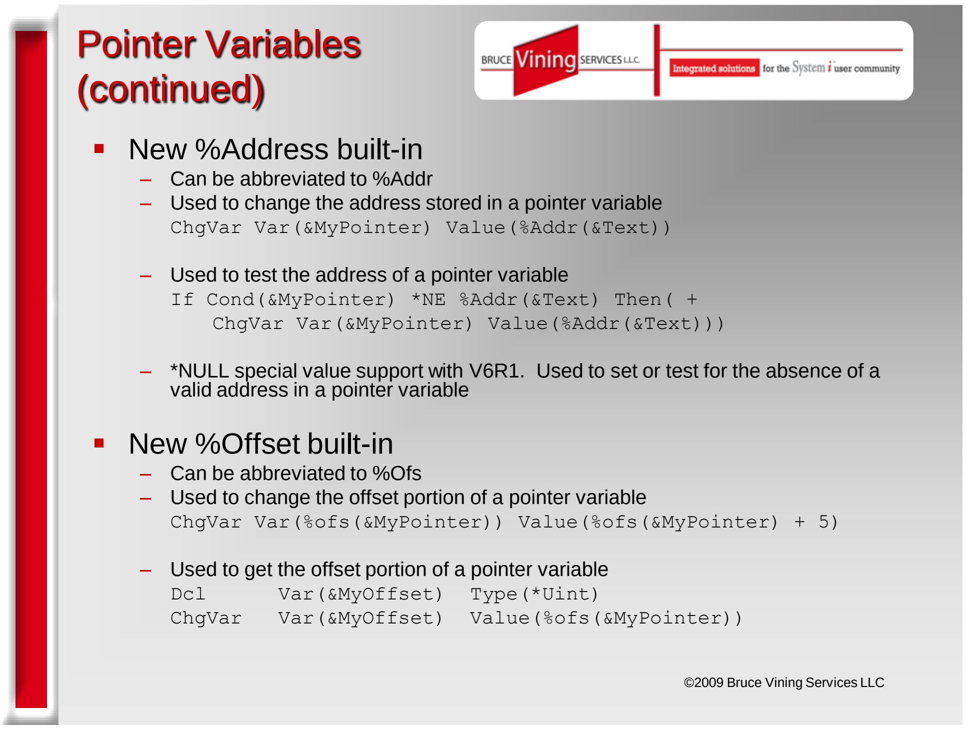
- New optional Address keyword for Dcl command

```
Dcl     Var(&Text)        Type(*Char) Len(20) +
                            Value(‘Some text’)
Dcl     Var(&MyPointer) Type(*Ptr) Address(&Text 5)
```

Caution

  - Sets &MyPointer to the address of the 6th byte of CL variable &Text (the initial ‘t’ of ‘text’)
  - The offset (5 in the example) is optional and defaults to 0 (the start of the variable) *Note that offset is base 0*

- Dcl command restrictions/considerations
  - Len keyword is not valid if Type(*Ptr). Pointers are fixed at 16 bytes in length
  - Value keyword is not valid if Type(*Ptr). The Address keyword is used to set the initial address assigned to the pointer variable



Pointer Variables (continued)

- New %Address built-in
  - Can be abbreviated to %Addr
  - Used to change the address stored in a pointer variable

    ```
    ChgVar Var(&MyPointer) Value(%Addr(&Text))
    ```

  - Used to test the address of a pointer variable

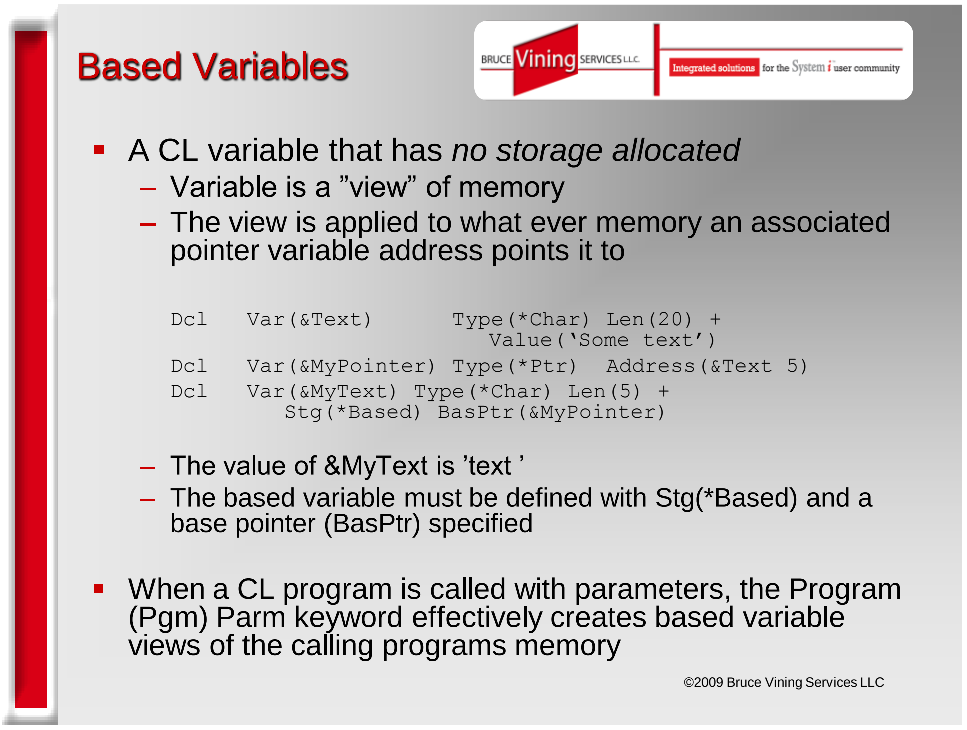
    ```
    If Cond(&MyPointer) *NE %Addr(&Text) Then( +
        ChgVar Var(&MyPointer) Value(%Addr(&Text)))
    ```

  - *NULL special value support with V6R1. Used to set or test for the absence of a valid address in a pointer variable
- New %Offset built-in
  - Can be abbreviated to %Ofs
  - Used to change the offset portion of a pointer variable

    ```
    ChgVar Var(%ofs(&MyPointer)) Value(%ofs(&MyPointer) + 5)
    ```

  - Used to get the offset portion of a pointer variable

    ```
    Dcl       Var(&MyOffset)   Type(*Uint)
    ChgVar    Var(&MyOffset)   Value(%ofs(&MyPointer))
    ```

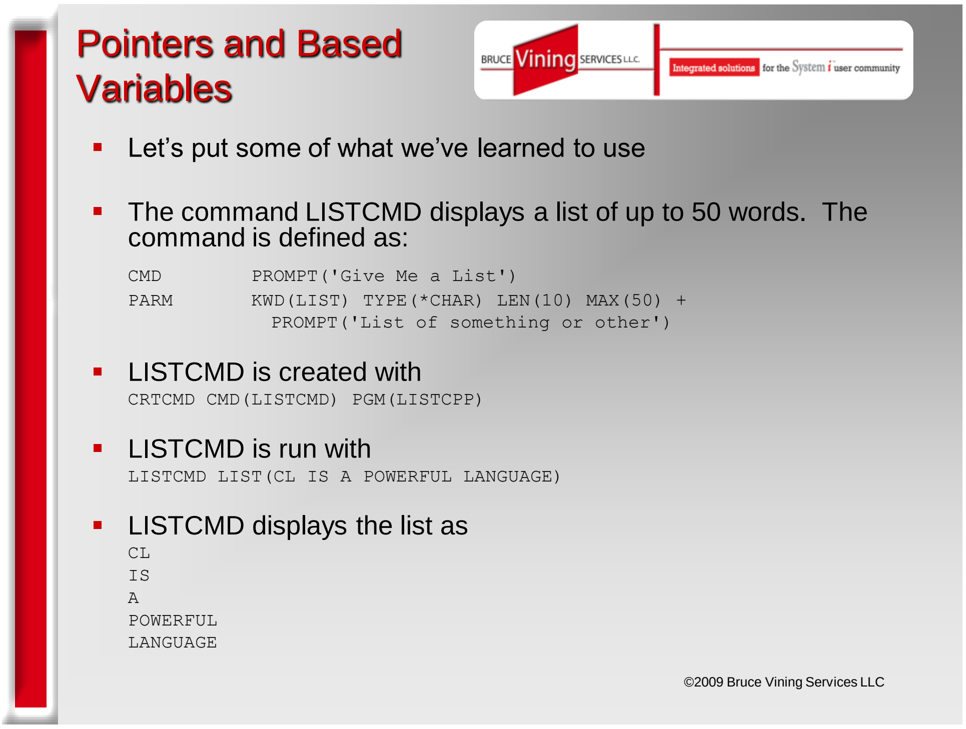


Based Variables

- A CL variable that has *no storage allocated*
  - Variable is a ”view” of memory
  - The view is applied to what ever memory an associated pointer variable address points it to

    ```
    Dcl    Var(&Text)       Type(*Char) Len(20) +
                               Value(‘Some text’)
    Dcl    Var(&MyPointer) Type(*Ptr)   Address(&Text 5)
    Dcl    Var(&MyText) Type(*Char) Len(5) +
              Stg(*Based) BasPtr(&MyPointer)
    ```

  - The value of &MyText is ’text ’
  - The based variable must be defined with Stg(*Based) and a base pointer (BasPtr) specified
- When a CL program is called with parameters, the Program (Pgm) Parm keyword effectively creates based variable views of the calling programs memory



Pointers and Based Variables

- Let's put some of what we've learned to use
- The command LISTCMD displays a list of up to 50 words. The command is defined as:

```
CMD        PROMPT('Give Me a List')
PARM       KWD(LIST) TYPE(*CHAR) LEN(10) MAX(50) +
             PROMPT('List of something or other')
```

- LISTCMD is created with

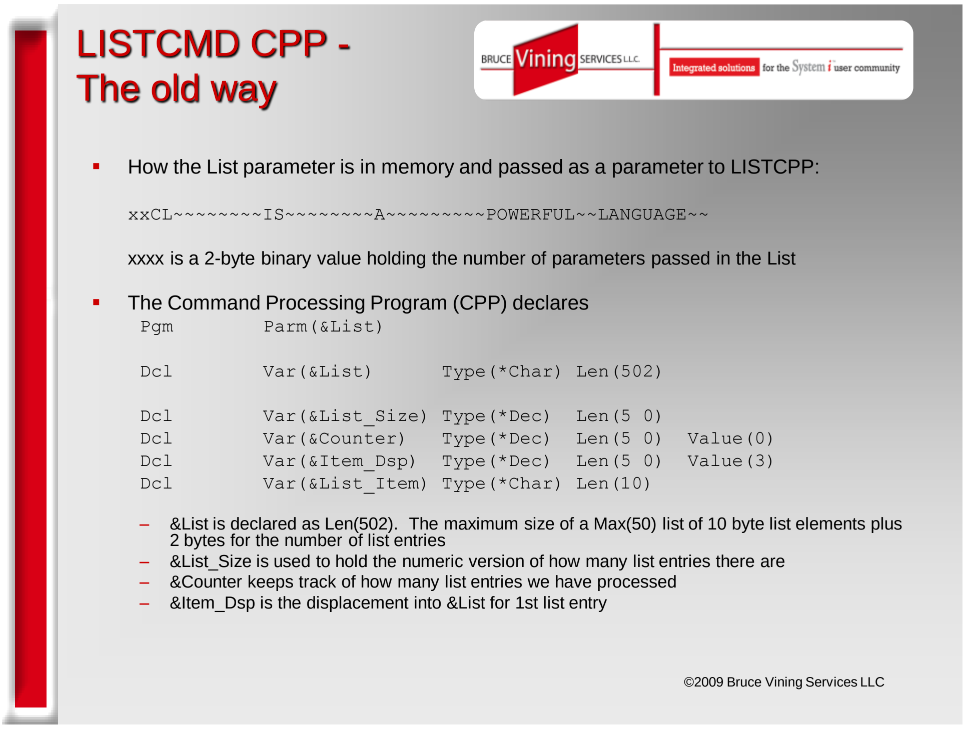
```
CRTCMD CMD(LISTCMD) PGM(LISTCPP)
```

- LISTCMD is run with

```
LISTCMD LIST(CL IS A POWERFUL LANGUAGE)
```

- LISTCMD displays the list as

```
CL
IS
A
POWERFUL
LANGUAGE
```



LISTCMD CPP - The old way

- How the List parameter is in memory and passed as a parameter to LISTCPP:

```
xxCL~~~~~~~~IS~~~~~~~~A~~~~~~~~~POWERFUL~~LANGUAGE~~
```

  xxxx is a 2-byte binary value holding the number of parameters passed in the List

- The Command Processing Program (CPP) declares

```
 Pgm        Parm(&List)

 Dcl        Var(&List)      Type(*Char) Len(502)

 Dcl        Var(&List_Size) Type(*Dec)  Len(5 0)
 Dcl        Var(&Counter)   Type(*Dec)  Len(5 0)  Value(0)
 Dcl        Var(&Item_Dsp)  Type(*Dec)  Len(5 0)  Value(3)
 Dcl        Var(&List_Item) Type(*Char) Len(10)
```

  - &List is declared as Len(502). The maximum size of a Max(50) list of 10 byte list elements plus 2 bytes for the number of list entries
  - &List_Size is used to hold the numeric version of how many list entries there are
  - &Counter keeps track of how many list entries we have processed
  - &Item_Dsp is the displacement into &List for 1st list entry

LISTCMD CPP - The old way

- How the List parameter is in memory:

```
xxCL~~~~~~~~~IS~~~~~~~~~A~~~~~~~~~~POWERFUL~~LANGUAGE~~
```

- The CPP logic

```
             ChgVar     Var(&List_Size) Value(%Bin(&List 1 2))
 Again:      If         Cond(&Counter *LT &List_Size) Then(Do)
                         ChgVar Var(&List_Item) +
                           Value(%Sst(&List &Item_Dsp 10))
                         SndPgmMsg Msg(&List_Item)
                         ChgVar Var(&Item_Dsp) Value(&Item_Dsp + 10)
                         ChgVar Var(&Counter)  Value(&Counter + 1)
                         GoTo CmdLbl(Again)
                         EndDo
```

  – Get the number of list entries using the %Bin builtin and convert it to a numeric value (&List_Size)

LISTCMD CPP - The old way

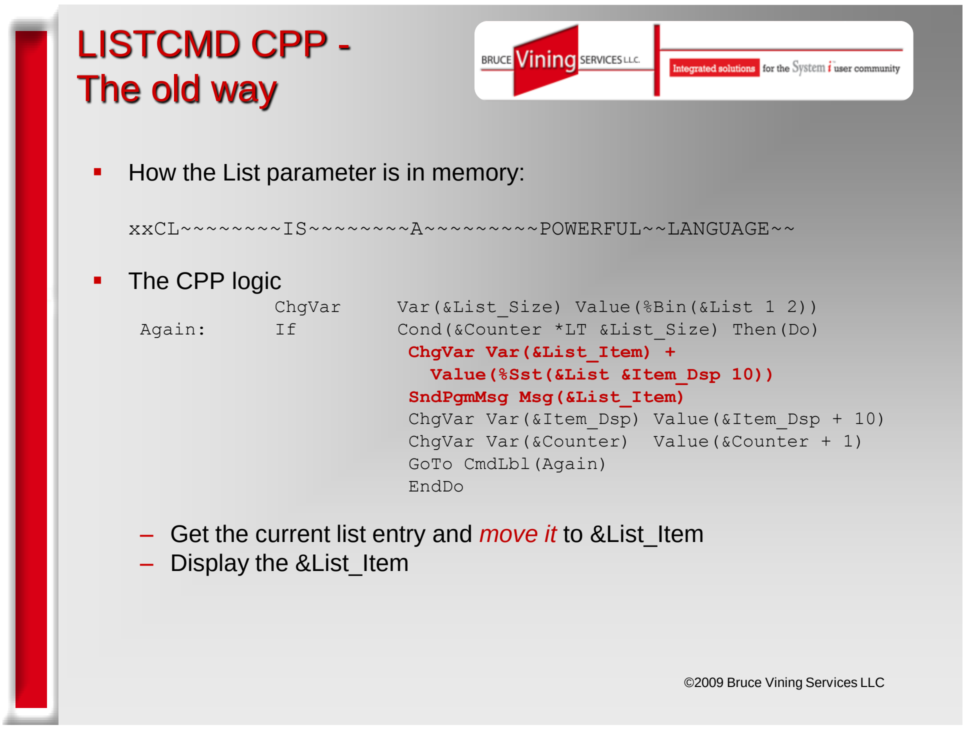
- How the List parameter is in memory:

```
xxCL~~~~~~~~~IS~~~~~~~~~A~~~~~~~~~~POWERFUL~~LANGUAGE~~
```

- The CPP logic

```
             ChgVar     Var(&List_Size) Value(%Bin(&List 1 2))
 Again:      If         Cond(&Counter *LT &List_Size) Then(Do)
                         ChgVar Var(&List_Item) +
                           Value(%Sst(&List &Item_Dsp 10))
                         SndPgmMsg Msg(&List_Item)
                         ChgVar Var(&Item_Dsp) Value(&Item_Dsp + 10)
                         ChgVar Var(&Counter)  Value(&Counter + 1)
                         GoTo CmdLbl(Again)
                         EndDo
```

  – Check if all list entires have been processed
  – If not run the Do loop
  – If all have been processed continue processing after the EndDo



LISTCMD CPP - The old way

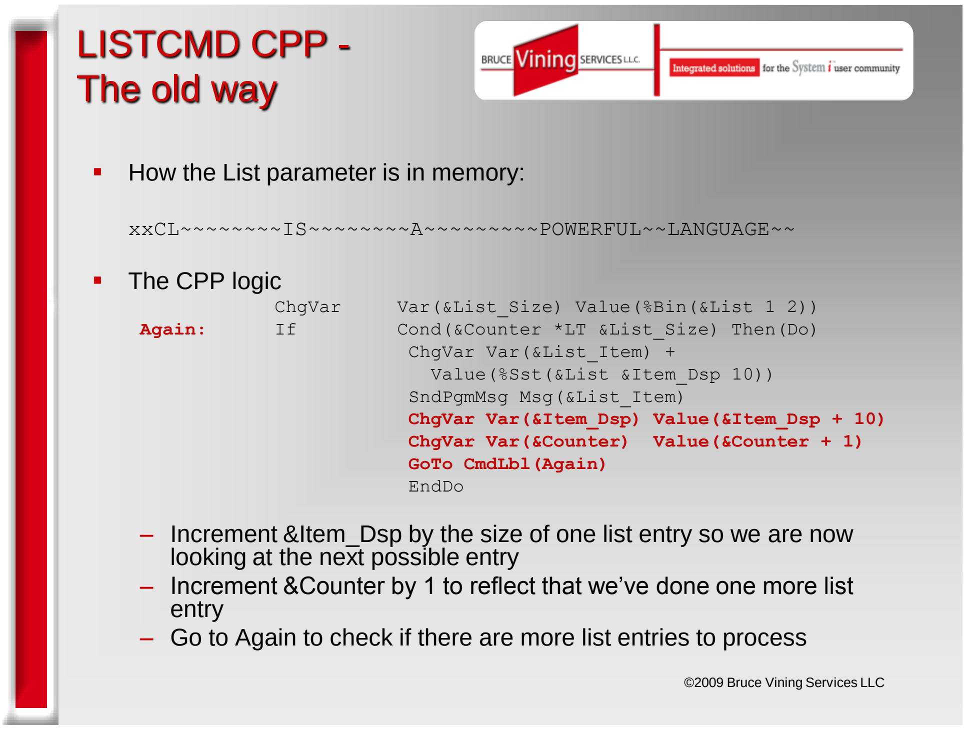
- How the List parameter is in memory:

```
xxCL~~~~~~~~~IS~~~~~~~~~A~~~~~~~~~~POWERFUL~~LANGUAGE~~
```

- The CPP logic

```
             ChgVar     Var(&List_Size) Value(%Bin(&List 1 2))
Again:       If         Cond(&Counter *LT &List_Size) Then(Do)
                         ChgVar Var(&List_Item) +
                           Value(%Sst(&List &Item_Dsp 10))
                         SndPgmMsg Msg(&List_Item)
                         ChgVar Var(&Item_Dsp) Value(&Item_Dsp + 10)
                         ChgVar Var(&Counter)  Value(&Counter + 1)
                         GoTo CmdLbl(Again)
                         EndDo
```

  – Get the current list entry and *move it* to &List_Item
  – Display the &List_Item



LISTCMD CPP - The old way

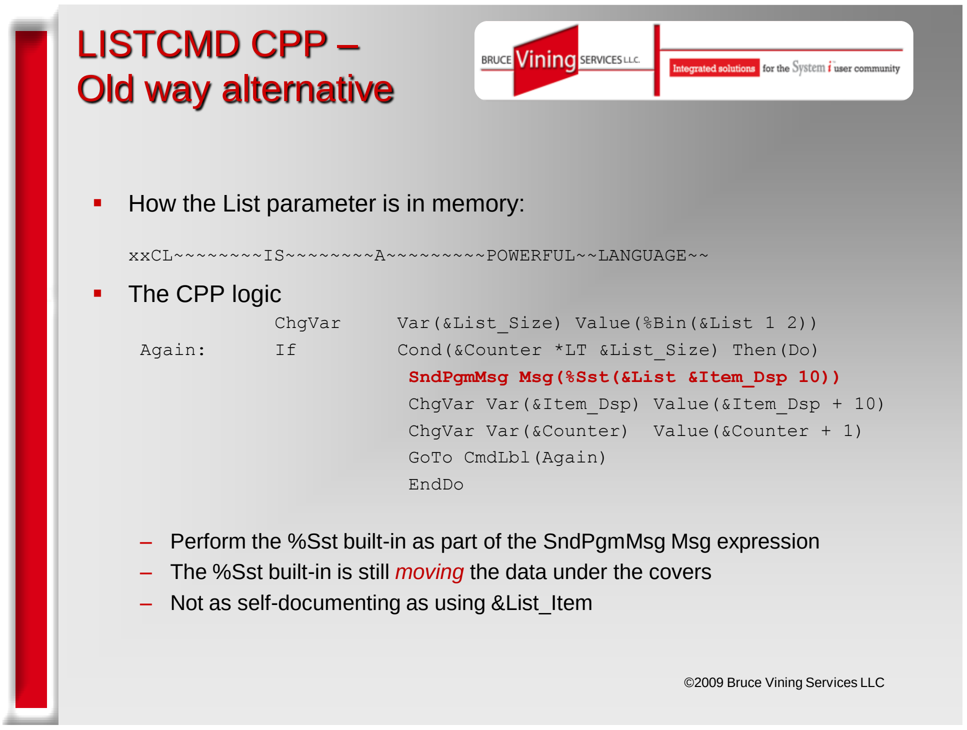
- How the List parameter is in memory:

```
xxCL~~~~~~~~~IS~~~~~~~~~A~~~~~~~~~~POWERFUL~~LANGUAGE~~
```

- The CPP logic

```
             ChgVar     Var(&List_Size) Value(%Bin(&List 1 2))
Again:       If         Cond(&Counter *LT &List_Size) Then(Do)
                         ChgVar Var(&List_Item) +
                           Value(%Sst(&List &Item_Dsp 10))
                         SndPgmMsg Msg(&List_Item)
                         ChgVar Var(&Item_Dsp) Value(&Item_Dsp + 10)
                         ChgVar Var(&Counter)  Value(&Counter + 1)
                         GoTo CmdLbl(Again)
                         EndDo
```

  – Increment &Item_Dsp by the size of one list entry so we are now looking at the next possible entry
  – Increment &Counter by 1 to reflect that we've done one more list entry
  – Go to Again to check if there are more list entries to process



LISTCMD CPP – Old way alternative

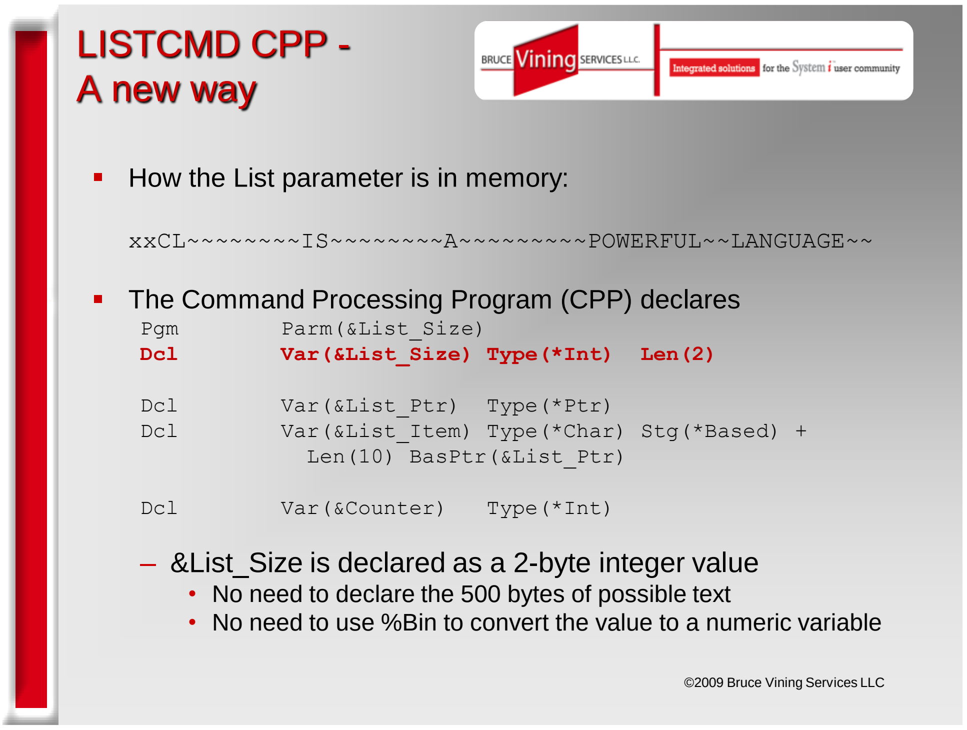
- How the List parameter is in memory:

```
xxCL~~~~~~~~IS~~~~~~~~A~~~~~~~~~POWERFUL~~LANGUAGE~~
```

- The CPP logic

```
             ChgVar     Var(&List_Size) Value(%Bin(&List 1 2))
Again:       If         Cond(&Counter *LT &List_Size) Then(Do)
                         SndPgmMsg Msg(%Sst(&List &Item_Dsp 10))
                         ChgVar Var(&Item_Dsp) Value(&Item_Dsp + 10)
                         ChgVar Var(&Counter)  Value(&Counter + 1)
                         GoTo CmdLbl(Again)
                         EndDo
```

  – Perform the %Sst built-in as part of the SndPgmMsg Msg expression
  – The %Sst built-in is still *moving* the data under the covers
  – Not as self-documenting as using &List_Item



LISTCMD CPP - A new way

- How the List parameter is in memory:

```
xxCL~~~~~~~~IS~~~~~~~~A~~~~~~~~~POWERFUL~~LANGUAGE~~
```

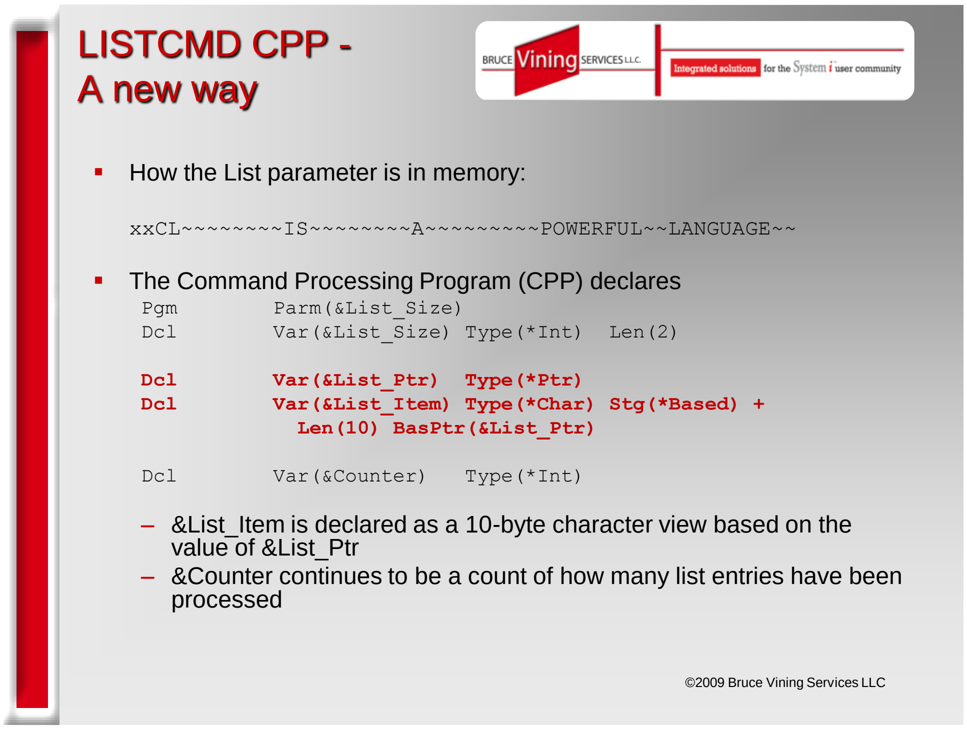
- The Command Processing Program (CPP) declares

```
Pgm          Parm(&List_Size)
Dcl          Var(&List_Size) Type(*Int)   Len(2)

Dcl          Var(&List_Ptr)  Type(*Ptr)
Dcl          Var(&List_Item) Type(*Char) Stg(*Based) +
               Len(10) BasPtr(&List_Ptr)

Dcl          Var(&Counter)   Type(*Int)
```

  – &List_Size is declared as a 2-byte integer value
    - No need to declare the 500 bytes of possible text
    - No need to use %Bin to convert the value to a numeric variable



LISTCMD CPP - A new way

- How the List parameter is in memory:

```
xxCL~~~~~~~~~IS~~~~~~~~~A~~~~~~~~~~POWERFUL~~LANGUAGE~~
```

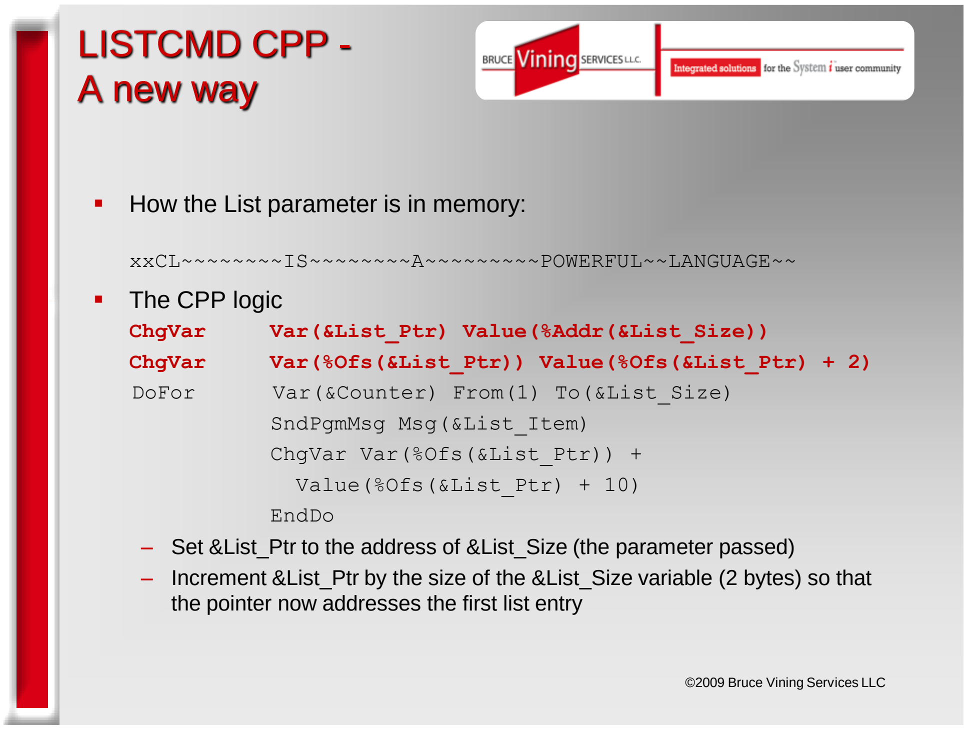
- The Command Processing Program (CPP) declares

```
Pgm        Parm(&List_Size)
Dcl        Var(&List_Size) Type(*Int)  Len(2)

Dcl        Var(&List_Ptr)  Type(*Ptr)
Dcl        Var(&List_Item) Type(*Char) Stg(*Based) +
             Len(10) BasPtr(&List_Ptr)

Dcl        Var(&Counter)   Type(*Int)
```

  – &List_Item is declared as a 10-byte character view based on the value of &List_Ptr
  – &Counter continues to be a count of how many list entries have been processed



LISTCMD CPP - A new way

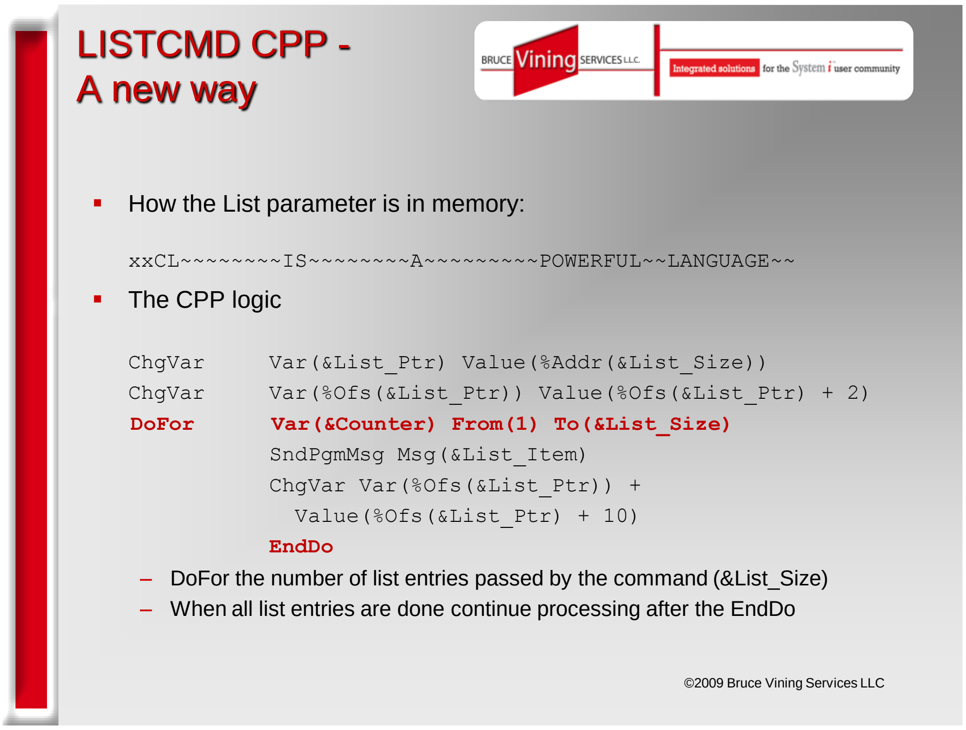
- How the List parameter is in memory:

```
xxCL~~~~~~~~~IS~~~~~~~~~A~~~~~~~~~~POWERFUL~~LANGUAGE~~
```

- The CPP logic

```
ChgVar     Var(&List_Ptr) Value(%Addr(&List_Size))
ChgVar     Var(%Ofs(&List_Ptr)) Value(%Ofs(&List_Ptr) + 2)
DoFor      Var(&Counter) From(1) To(&List_Size)
           SndPgmMsg Msg(&List_Item)
           ChgVar Var(%Ofs(&List_Ptr)) +
             Value(%Ofs(&List_Ptr) + 10)
           EndDo
```

  – Set &List_Ptr to the address of &List_Size (the parameter passed)
  – Increment &List_Ptr by the size of the &List_Size variable (2 bytes) so that the pointer now addresses the first list entry



LISTCMD CPP - A new way

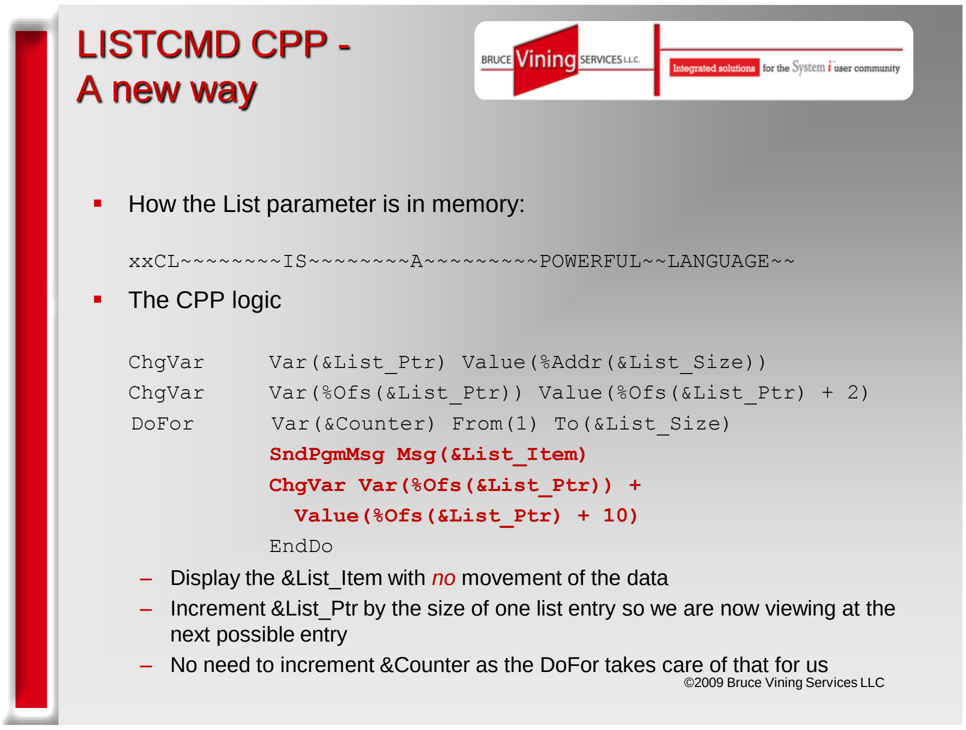
- How the List parameter is in memory:

```
xxCL~~~~~~~~~IS~~~~~~~~~A~~~~~~~~~~POWERFUL~~LANGUAGE~~
```

- The CPP logic

```
ChgVar      Var(&List_Ptr) Value(%Addr(&List_Size))
ChgVar      Var(%Ofs(&List_Ptr)) Value(%Ofs(&List_Ptr) + 2)
DoFor       Var(&Counter) From(1) To(&List_Size)
            SndPgmMsg Msg(&List_Item)
            ChgVar Var(%Ofs(&List_Ptr)) +
              Value(%Ofs(&List_Ptr) + 10)
            EndDo
```

  – DoFor the number of list entries passed by the command (&List_Size)
  – When all list entries are done continue processing after the EndDo



LISTCMD CPP - A new way

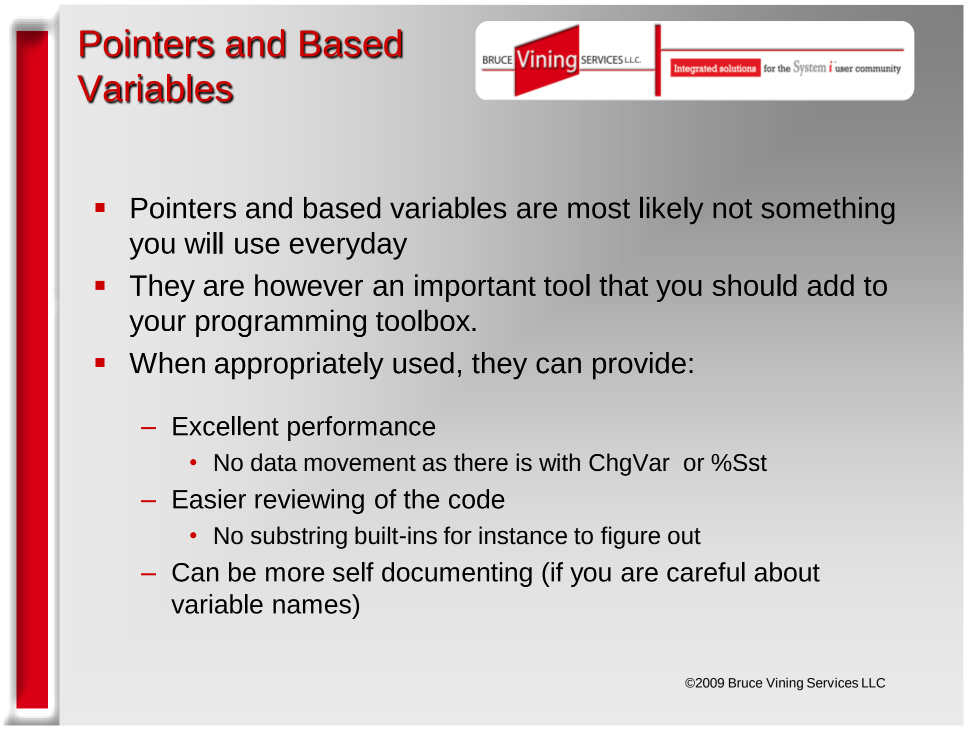
- How the List parameter is in memory:

```
xxCL~~~~~~~~~IS~~~~~~~~~A~~~~~~~~~~POWERFUL~~LANGUAGE~~
```

- The CPP logic

```
ChgVar      Var(&List_Ptr) Value(%Addr(&List_Size))
ChgVar      Var(%Ofs(&List_Ptr)) Value(%Ofs(&List_Ptr) + 2)
DoFor       Var(&Counter) From(1) To(&List_Size)
            SndPgmMsg Msg(&List_Item)
            ChgVar Var(%Ofs(&List_Ptr)) +
              Value(%Ofs(&List_Ptr) + 10)
            EndDo
```

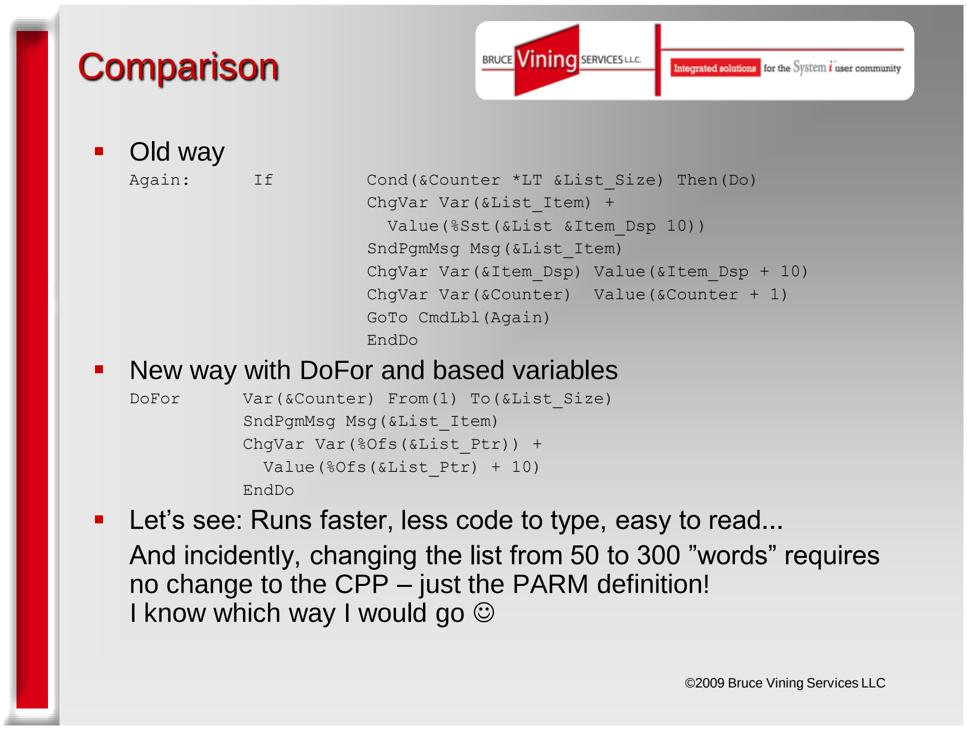
  – Display the &List_Item with *no* movement of the data
  – Increment &List_Ptr by the size of one list entry so we are now viewing at the next possible entry
  – No need to increment &Counter as the DoFor takes care of that for us



Pointers and Based Variables

BRUCE Vining SERVICES LLC — Integrated solutions for the System i user community

- Pointers and based variables are most likely not something you will use everyday
- They are however an important tool that you should add to your programming toolbox.
- When appropriately used, they can provide:
  - Excellent performance
    - No data movement as there is with ChgVar or %Sst
  - Easier reviewing of the code
    - No substring built-ins for instance to figure out
  - Can be more self documenting (if you are careful about variable names)



Comparison

BRUCE Vining SERVICES LLC — Integrated solutions for the System i user community

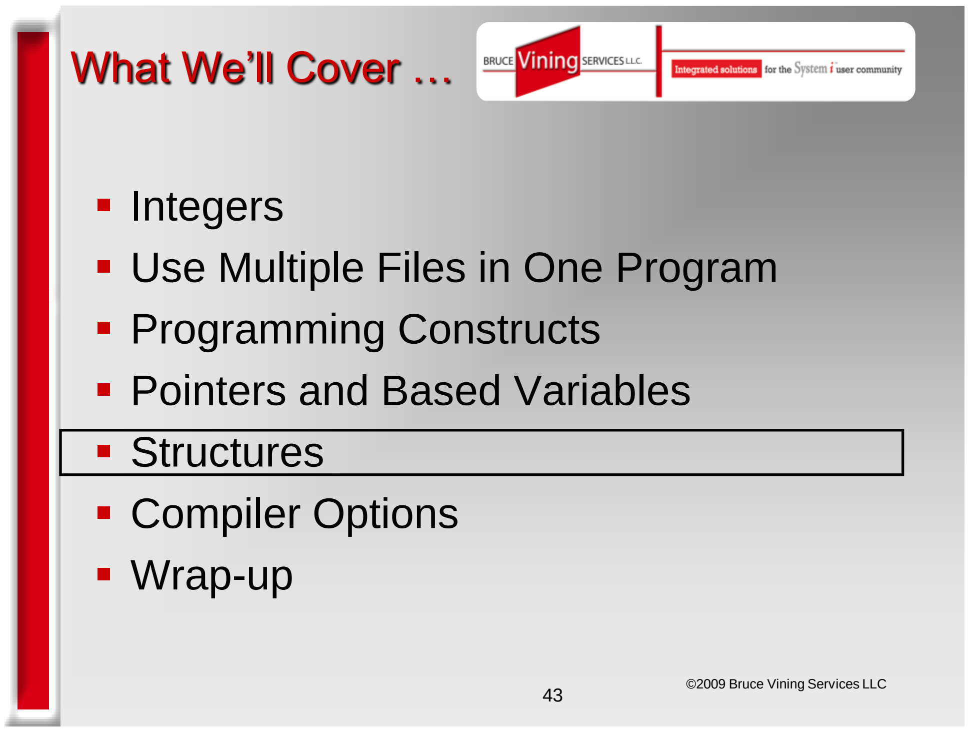
- Old way

```
Again:      If          Cond(&Counter *LT &List_Size) Then(Do)
                        ChgVar Var(&List_Item) +
                          Value(%Sst(&List &Item_Dsp 10))
                        SndPgmMsg Msg(&List_Item)
                        ChgVar Var(&Item_Dsp) Value(&Item_Dsp + 10)
                        ChgVar Var(&Counter)  Value(&Counter + 1)
                        GoTo CmdLbl(Again)
                        EndDo
```

- New way with DoFor and based variables

```
DoFor      Var(&Counter) From(1) To(&List_Size)
           SndPgmMsg Msg(&List_Item)
           ChgVar Var(%Ofs(&List_Ptr)) +
             Value(%Ofs(&List_Ptr) + 10)
           EndDo
```

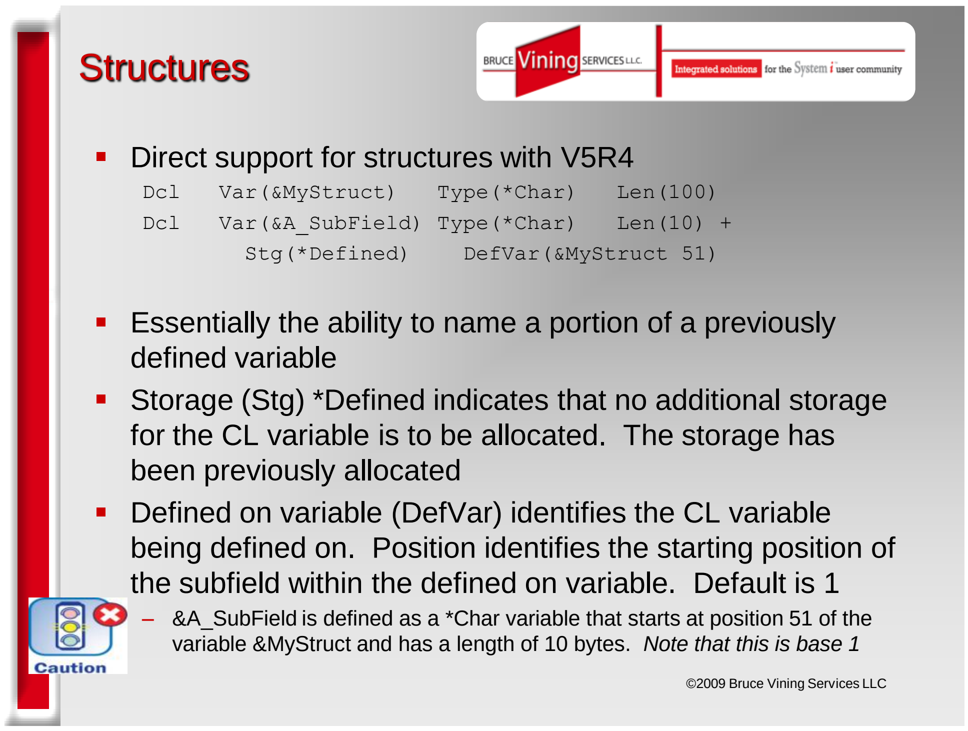
- Let’s see: Runs faster, less code to type, easy to read...
  And incidently, changing the list from 50 to 300 ”words” requires no change to the CPP – just the PARM definition!
  I know which way I would go ☺



What We'll Cover …

- Integers
- Use Multiple Files in One Program
- Programming Constructs
- Pointers and Based Variables
- Structures
- Compiler Options
- Wrap-up

43



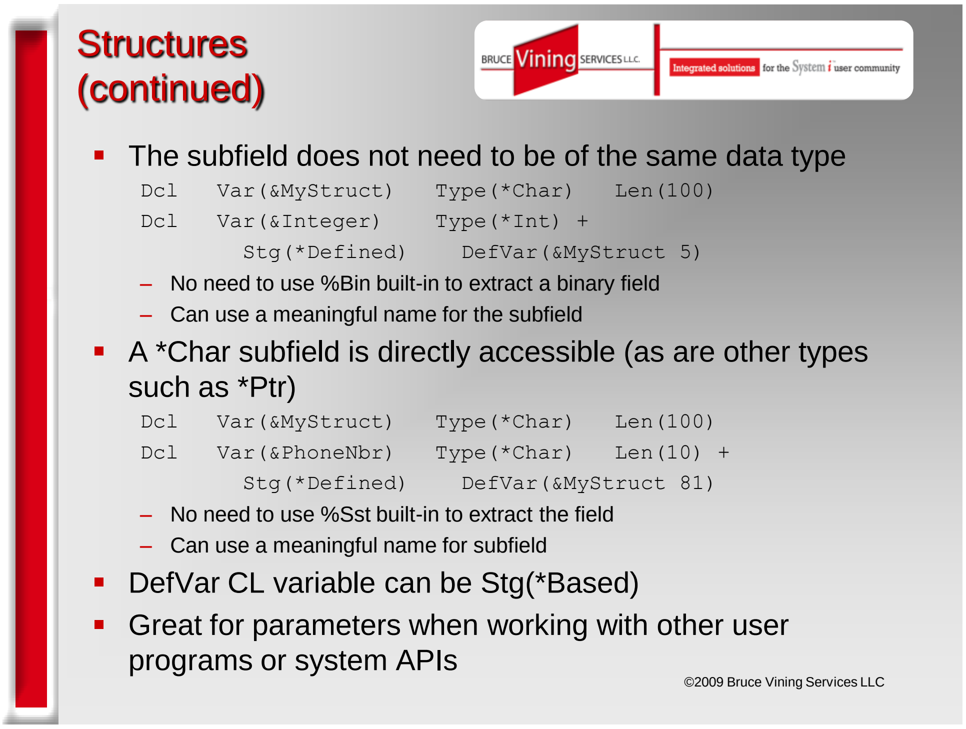
Structures

- Direct support for structures with V5R4

```
Dcl    Var(&MyStruct)    Type(*Char)    Len(100)
Dcl    Var(&A_SubField)  Type(*Char)    Len(10) +
         Stg(*Defined)     DefVar(&MyStruct 51)
```

- Essentially the ability to name a portion of a previously defined variable
- Storage (Stg) *Defined indicates that no additional storage for the CL variable is to be allocated. The storage has been previously allocated
- Defined on variable (DefVar) identifies the CL variable being defined on. Position identifies the starting position of the subfield within the defined on variable. Default is 1
  - &A_SubField is defined as a *Char variable that starts at position 51 of the variable &MyStruct and has a length of 10 bytes. *Note that this is base 1*

Caution



Structures (continued)

- The subfield does not need to be of the same data type

```
Dcl   Var(&MyStruct)   Type(*Char)   Len(100)
Dcl   Var(&Integer)    Type(*Int) +
        Stg(*Defined)    DefVar(&MyStruct 5)
```

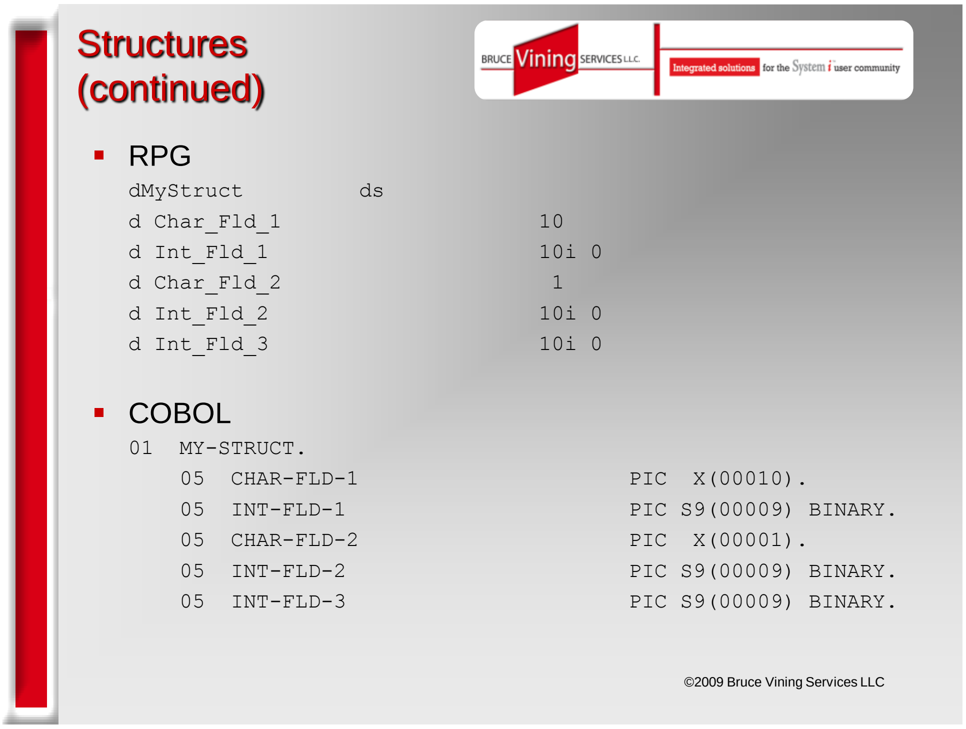
  – No need to use %Bin built-in to extract a binary field
  – Can use a meaningful name for the subfield

- A *Char subfield is directly accessible (as are other types such as *Ptr)

```
Dcl   Var(&MyStruct)   Type(*Char)   Len(100)
Dcl   Var(&PhoneNbr)   Type(*Char)   Len(10) +
        Stg(*Defined)    DefVar(&MyStruct 81)
```

  – No need to use %Sst built-in to extract the field
  – Can use a meaningful name for subfield

- DefVar CL variable can be Stg(*Based)
- Great for parameters when working with other user programs or system APIs



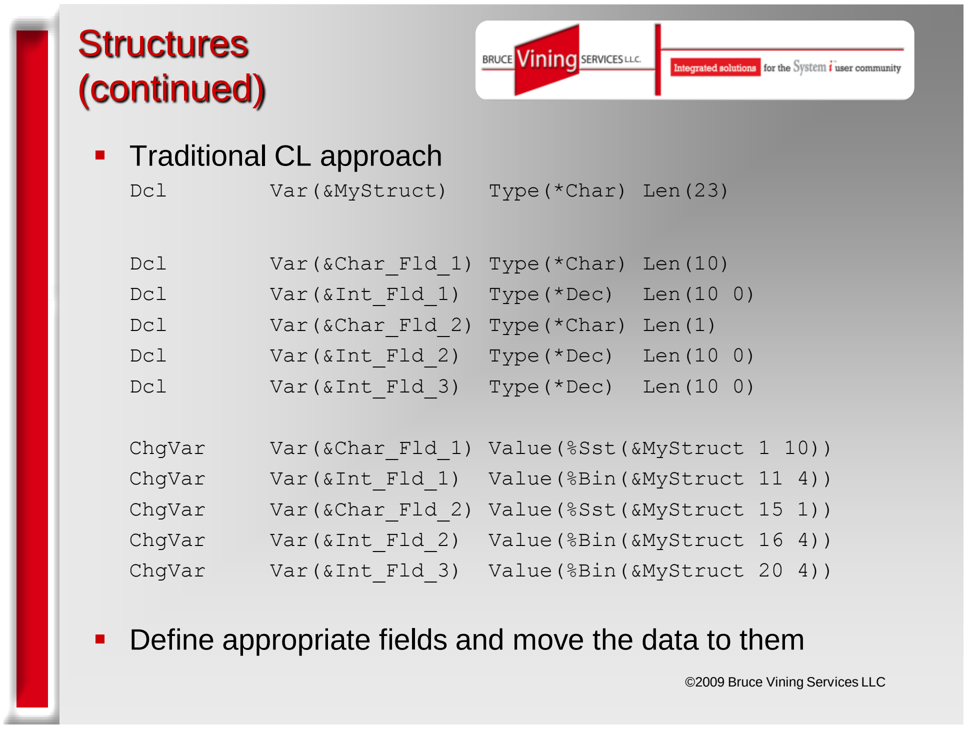
Structures (continued)

- RPG

```
dMyStruct         ds
d Char_Fld_1                    10
d Int_Fld_1                     10i 0
d Char_Fld_2                     1
d Int_Fld_2                     10i 0
d Int_Fld_3                     10i 0
```

- COBOL

```
01  MY-STRUCT.
    05  CHAR-FLD-1                  PIC  X(00010).
    05  INT-FLD-1                   PIC S9(00009) BINARY.
    05  CHAR-FLD-2                  PIC  X(00001).
    05  INT-FLD-2                   PIC S9(00009) BINARY.
    05  INT-FLD-3                   PIC S9(00009) BINARY.
```



Structures (continued)

BRUCE Vining SERVICES LLC Integrated solutions for the System *i* user community

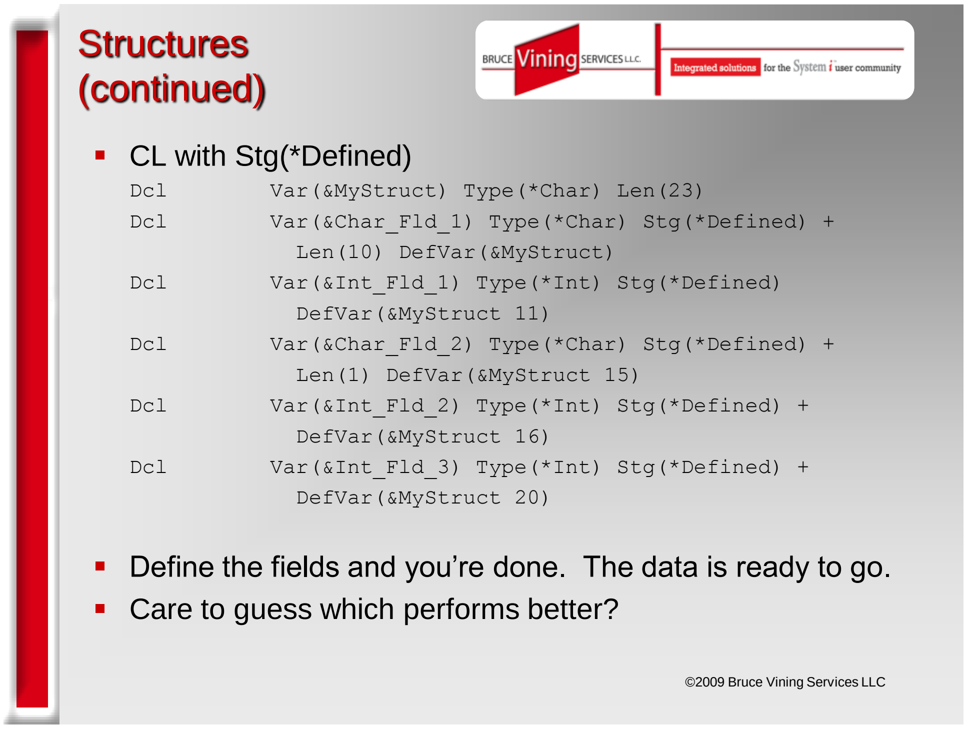
- Traditional CL approach

```
Dcl         Var(&MyStruct)    Type(*Char) Len(23)

Dcl         Var(&Char_Fld_1) Type(*Char) Len(10)
Dcl         Var(&Int_Fld_1)  Type(*Dec)  Len(10 0)
Dcl         Var(&Char_Fld_2) Type(*Char) Len(1)
Dcl         Var(&Int_Fld_2)  Type(*Dec)  Len(10 0)
Dcl         Var(&Int_Fld_3)  Type(*Dec)  Len(10 0)

ChgVar      Var(&Char_Fld_1) Value(%Sst(&MyStruct 1 10))
ChgVar      Var(&Int_Fld_1)  Value(%Bin(&MyStruct 11 4))
ChgVar      Var(&Char_Fld_2) Value(%Sst(&MyStruct 15 1))
ChgVar      Var(&Int_Fld_2)  Value(%Bin(&MyStruct 16 4))
ChgVar      Var(&Int_Fld_3)  Value(%Bin(&MyStruct 20 4))
```

- Define appropriate fields and move the data to them



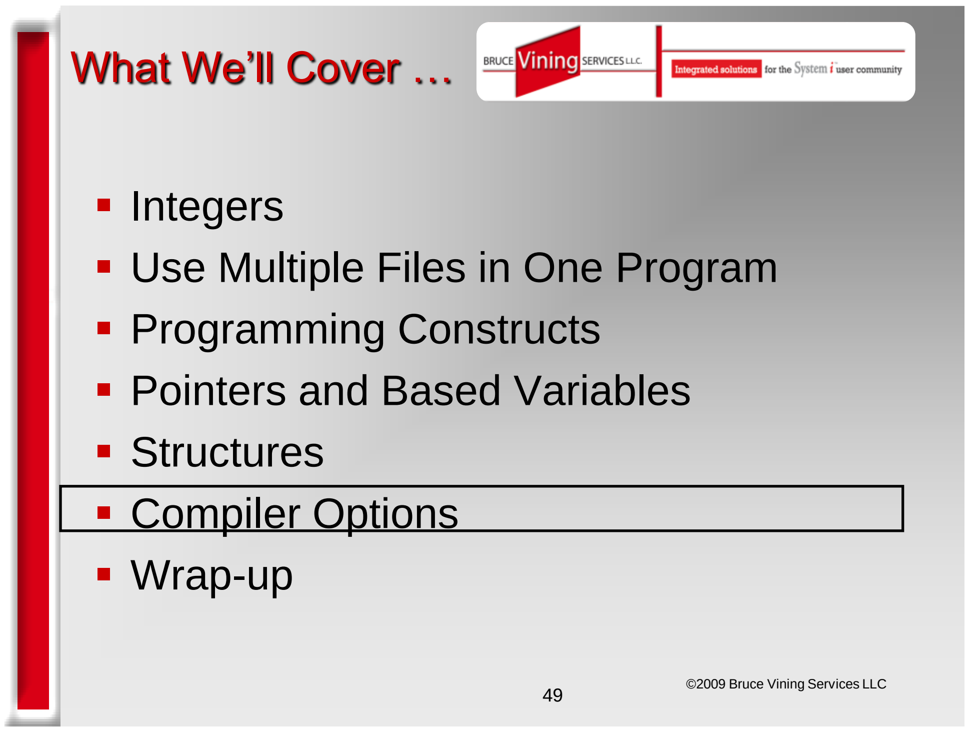
Structures (continued)

BRUCE Vining SERVICES LLC Integrated solutions for the System *i* user community

- CL with Stg(*Defined)

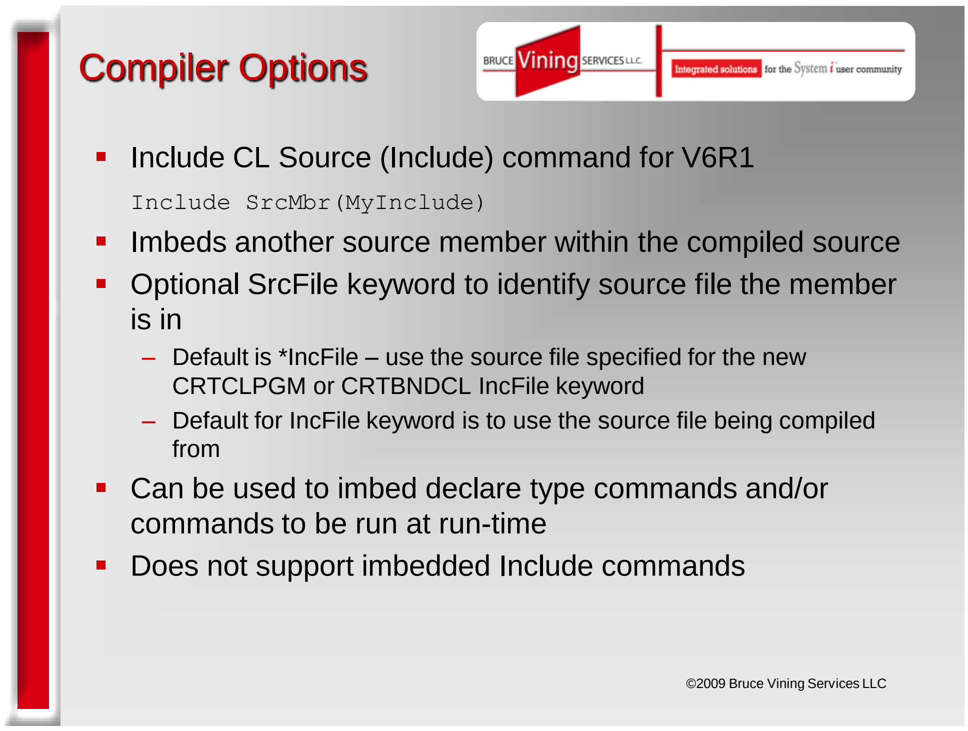
```
Dcl         Var(&MyStruct) Type(*Char) Len(23)
Dcl         Var(&Char_Fld_1) Type(*Char) Stg(*Defined) +
              Len(10) DefVar(&MyStruct)
Dcl         Var(&Int_Fld_1) Type(*Int) Stg(*Defined)
              DefVar(&MyStruct 11)
Dcl         Var(&Char_Fld_2) Type(*Char) Stg(*Defined) +
              Len(1) DefVar(&MyStruct 15)
Dcl         Var(&Int_Fld_2) Type(*Int) Stg(*Defined) +
              DefVar(&MyStruct 16)
Dcl         Var(&Int_Fld_3) Type(*Int) Stg(*Defined) +
              DefVar(&MyStruct 20)
```

- Define the fields and you’re done. The data is ready to go.
- Care to guess which performs better?



What We'll Cover …

- Integers
- Use Multiple Files in One Program
- Programming Constructs
- Pointers and Based Variables
- Structures
- Compiler Options
- Wrap-up



Compiler Options

- Include CL Source (Include) command for V6R1

```
Include SrcMbr(MyInclude)
```

- Imbeds another source member within the compiled source
- Optional SrcFile keyword to identify source file the member is in
  - Default is *IncFile – use the source file specified for the new CRTCLPGM or CRTBNDCL IncFile keyword
  - Default for IncFile keyword is to use the source file being compiled from
- Can be used to imbed declare type commands and/or commands to be run at run-time
- Does not support imbedded Include commands

Compiler Options

- Declare Processing Options (DclPrcOpt) command in the CL source member can define additional compiler processing options with V6R1

```
DclPrcOpt UsrPrf(*Owner) BndDir(MyBndDir) etc.
```

- Options supported:

```
ActGrp         Log
AlwRtvSrc      SrtSeq
Aut            StgMdl
BndDir         Text
BndSrvPgm      UsrPrf
DftActGrp
LangID
```

- DclPrcOpt value takes precedence over Crt command
- Avoid lengthy problem determination due to a program being compiled with the wrong options

Tip

What We'll Cover …

- Integers
- Use Multiple Files in One Program
- Programming Constructs
- Pointers and Based Variables
- Structures
- Compile Options
- Wrap-up

Additional Resources

- IBM i5/OS Information Center
  - V5R3: http://publib.boulder.ibm.com/infocenter/iseries/v5r3/index.jsp
  - V5R4: http://publib.boulder.ibm.com/infocenter/iseries/v5r4/index.jsp
  - V6R1: http://publib.boulder.ibm.com/infocenter/systems/scope/i5os/index.jsp
- Many examples in my CL-related articles
  - http://www.brucevining.com/
  - Select Publications
  - Select Control Language (CL)

Some Key Points to Take Home

- CL continues to grow more flexible and powerful
- Recent CL enhancements can improve both your productivity and system performance – a true win/win situation ☺
  - Stg(*Defined)
  - Stg(*Based)
  - DoFor, DoWhile, DoUntil
  - DclPrcOpt – avoids mistakes when compiling…
- CL will continue to grow in the future

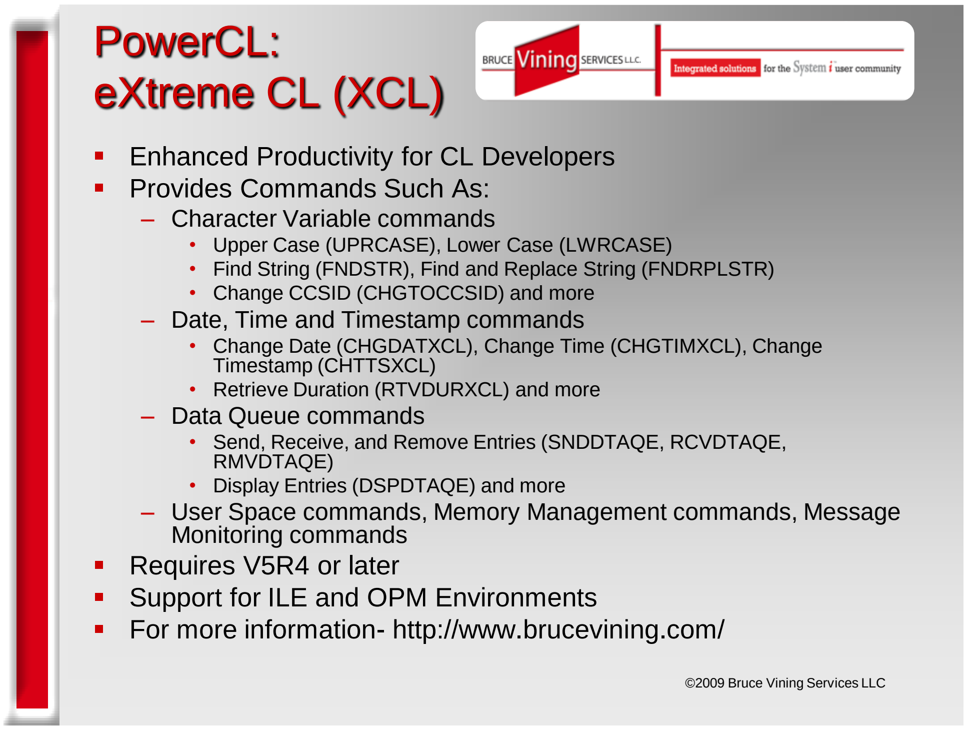
Future Possible Enhancements

- Support for 8-byte *Int and *Uint data types
- Encrypted source debug listing support
- RtvCLSrc support for ILE CL
- Higher precision *Dec support
- Arrays
- Date, Time, and Timestamp support
- Floating point support

- *But NO Guarantees*

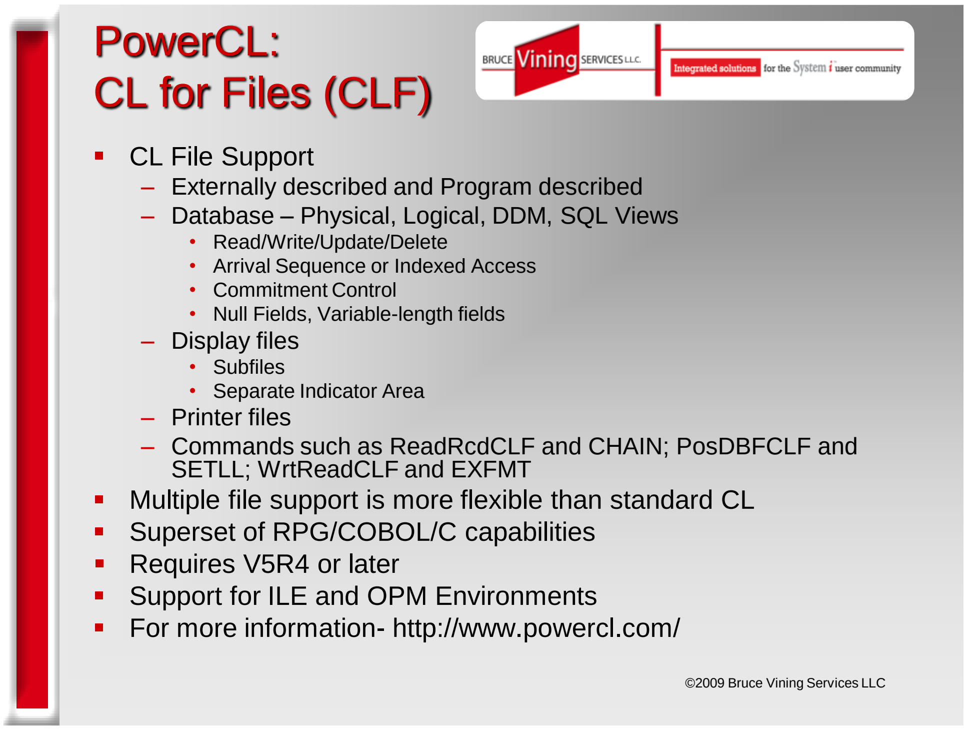
Last Chance Before the Break

How to contact me:
Bruce Vining
bvining@brucevining.com



PowerCL: eXtreme CL (XCL)

- Enhanced Productivity for CL Developers
- Provides Commands Such As:
  - Character Variable commands
    - Upper Case (UPRCASE), Lower Case (LWRCASE)
    - Find String (FNDSTR), Find and Replace String (FNDRPLSTR)
    - Change CCSID (CHGTOCCSID) and more
  - Date, Time and Timestamp commands
    - Change Date (CHGDATXCL), Change Time (CHGTIMXCL), Change Timestamp (CHTTSXCL)
    - Retrieve Duration (RTVDURXCL) and more
  - Data Queue commands
    - Send, Receive, and Remove Entries (SNDDTAQE, RCVDTAQE, RMVDTAQE)
    - Display Entries (DSPDTAQE) and more
  - User Space commands, Memory Management commands, Message Monitoring commands
- Requires V5R4 or later
- Support for ILE and OPM Environments
- For more information- http://www.brucevining.com/



PowerCL: CL for Files (CLF)

- CL File Support
  - Externally described and Program described
  - Database – Physical, Logical, DDM, SQL Views
    - Read/Write/Update/Delete
    - Arrival Sequence or Indexed Access
    - Commitment Control
    - Null Fields, Variable-length fields
  - Display files
    - Subfiles
    - Separate Indicator Area
  - Printer files
  - Commands such as ReadRcdCLF and CHAIN; PosDBFCLF and SETLL; WrtReadCLF and EXFMT
- Multiple file support is more flexible than standard CL
- Superset of RPG/COBOL/C capabilities
- Requires V5R4 or later
- Support for ILE and OPM Environments
- For more information- http://www.powercl.com/