

# 4

## ***BASIC OPERATORS AND FUNCTIONS***

---

Chapter 3 covers the use of isolated constants and variables in a CL procedure. This chapter explains the various manipulations you can perform in CL to combine two or more constant or variable values.

### **THE CHGVAR COMMAND**

The Change Variable (CHGVAR) command is the primary means of assigning a new value to a variable. In this respect, CHGVAR is equivalent to the assignment statement found in other programming languages. Most programming languages use the equal sign (=) to this effect (Pascal uses the composite symbol := and RPG and COBOL use various verbs such as MOVE, COMPUTE, EVAL, and Z-ADD). The CHGVAR command has only two parameters, which are both mandatory:

- VAR: The name of the variable being changed.
- VALUE: The new value being assigned to the variable.

In the statement shown in Figure 4.1, variable &NUMBER is assigned the value 25. Whatever value was stored in &NUMBER is lost.

```
CHGVAR VAR(&NUMBER) VALUE(25)
```

*Figure 4.1: Assigning a value of 25 to the variable &NUMBER.*

VALUE can contain a single value or an expression. If it contains a single value (either a constant or another variable), the value is copied into the variable being changed. If VALUE contains an expression (such as  $10 + 5$ ), the expression is evaluated first, and the result is assigned to the variable.

If the variable is logical, VALUE must be a logical value of '0' (FALSE) or '1' (TRUE).

If the variable is character, VALUE can be character or decimal. Either way, the variable is cleared to all blanks before the new value is assigned. If the new value is a character value, the assignment is made directly. If the new value is a decimal value, the decimal value is first converted to character. Then the result is assigned to the variable.

If the variable is decimal, VALUE can be character or decimal. In either case, the variable is cleared to zero before the new value is assigned. If it is a character value, only the digits 0 to 9, an optional sign, and a decimal point or comma may be used. If it is a decimal value, the value is assigned to the variable with the decimal mark aligned.

Figure 4.2 shows a variety of examples of the CHGVAR command:

```
DCL &name      *CHAR  10
DCL &number    *DEC   (5 1)
DCL &toggle    *LGL   1

/* Character value to a character variable */
CHGVAR &name 'ABCDEFGHIJ'
      /* &NAME now contains 'ABCDEFGHIJ' */

CHGVAR &name 'X'
      /* &NAME now contains 'X'          ' */

CHGVAR &name 'AAABBBCCDDDEEE'
      /* &NAME now contains 'AAABBBCCCD' */

/* Decimal value to a character variable */
CHGVAR &name 25
      /* &NAME now contains '000000025' */
CHGVAR &name 123456789012345
      /* Produces error */

CHGVAR &name -32.5
      /* &NAME now contains '-0000032.5' */

/* Character value to decimal variable */
CHGVAR &number '12'
      /* &NUMBER now contains +0012.0 */

CHGVAR &number '72M'
      /* Produces error */

/* Decimal value to decimal variable */
CHGVAR &number 25.2
      /* &NUMBER now contains +0025.2 */

CHGVAR &number 3.14159265
      /* &NUMBER now contains +0003.1 */

CHGVAR &number -12
      /* &NUMBER now contains -0012.0 */

CHGVAR &number 6302931.77
      /* Produces error */

/* Logical value to logical variable */
CHGVAR &toggle '1'
      /* &TOGGLE has logical value TRUE */
```

Figure 4.2: Examples of the CHGVAR command.

## ARITHMETIC OPERATORS

The CHGVAR command also accepts arithmetic operations between decimal values. When arithmetic operations are used, the system calculates the result according to the standard mathematical rules, and assigns the result to the variable. In the example shown in Figure 4.3, variable &RESULT is assigned a value of 5.

```
CHGVAR VAR(&RESULT) VALUE(2 + 3)
```

Figure 4.3: Assigning a variable the value of a result of an arithmetic operation.

If you choose to omit keywords, you still have to use the parentheses around the expression as shown in Figure 4.4.

```
CHGVAR &result (2 + 3)
```

Figure 4.4: Assigning a variable the value of a result of an arithmetic operation (no keywords).

CL recognizes the four basic operations. See Table 4.1.

| <b>Table 4.1: Valid Arithmetic Operation in CL</b> |          |
|--|----------|
| +  | Add      |
| -  | Subtract |
| *  | Multiply |
| \  | Divide   |

In general, the compiler does not require you to leave blank spaces around the operators. The only exception is the division operator (/). Blank spaces are required for the division operator because the slash character is also used to separate the parts of a qualified name such as QGPL/QPRINT. To avoid problems and enforce consistency, you should leave a blank space around all arithmetic operators.

The VALUE parameter of the CHGVAR command can contain more than one operation, and even include parentheses to override the natural hierarchy of the operators (see Figure 4.5).

```
CHGVAR &result (2 * 3 + 4) /* Yields 10 */  
CHGVAR &result (2 * (3 + 4)) /* Yields 14 */
```

Figure 4.5: Examples of using multiple operations and parenthesis in a CHGVAR command.

## Substring Function

The substring function (coded either %SST or %SUBSTRING) can be used to extract a particular portion of a character variable. It is, therefore, equivalent to RPG's SUBST op code (RPG IV's %SUBST built-in function, too), or COBOL's reference modification. The %SST code requires three arguments:

- The name of the character variable from where a portion will be extracted.
- A decimal value representing the position, within the string, where the extraction will begin.
- Another decimal value representing the number of characters to extract.

Figure 4.6 shows two examples—one with keywords and one without.

```
CHGVAR VAR(&PORTION) VALUE(%SST(&LONG_NAME 12 5))  
CHGVAR &portion %SST(&long_name 12 5)
```

Figure 4.6: Examples of using %SST to assign a value to a variable.

Variable &PORTION will be assigned a five-byte string, containing characters 12, 13, 14, 15, and 16 of &LONG\_NAME. As shown in Figure 4.7, the same example would be valid using variables instead of the decimal constants 12 and 5.

```
CHGVAR &portion %SST(&long_name &start &length)
```

Figure 4.7: An example of using a variable to specify parameters for %SST.

In this example, both &START and &LENGTH must be decimal variables of whatever length is appropriate for the values they must contain. If &START is to contain a number such as 12, it could be a two-digit decimal variable or longer.

You may code the %SST function in the VAR parameter instead of in the VALUE parameter. When the function is coded this way, only a portion of the variable named in VAR will be changed. Figure 4.8 shows two examples of this.

```
CHGVAR &pattern 'XXX...XXX...'
      /* Assigns a pattern to variable &PATTERN */

CHGVAR %SST(&pattern 1 3) 'AAA'
      /* &PATTERN now contains 'AAA...XXX...' */
```

Figure 4.8: Examples of using %SST in the VAR parameter.

Instead of a variable name, you also can specify \*LDA (see Figure 4.9). In that case, the local data area for the job is retrieved (with %SST in the VALUE parameter) or changed (with %SST in the VAR parameter).

```
CHGVAR &a %SST(*LDA 1 10)
      /* Assigns &A the first 10 bytes of the LDA */

CHGVAR %SST(*LDA 101 8) ' '
      /* Changes LDA positions 101-108 to blanks */
```

Figure 4.9: Examples of \*LDA with %SST.

## CONCATENATING STRINGS

Character strings can be combined into longer strings with three built-in concatenation operations: \*CAT, \*BCAT, and \*TCAT. Table 4.2 explains their differences.

**Table 4.2: Explanation of Concatenation Operation**

| Operator    | Description  |
|-------------|--|
| *CAT or     | Joins the two strings exactly the way they are (including any trailing blanks). If the first string has three trailing blanks, the resulting string would have those three blanks in the middle.                   |
| *BCAT or  > | Joins the two strings in such a way that only one blank is placed between the two strings (even if trailing blanks are found in the first string). *BCAT guarantees that there will be one intervening blank only. |
| *TCAT or  < | Joins the two strings with the first string trimmed (without trailing blanks at all). *TCAT results in a string that has no embedded blanks, no matter how many trailing blanks were found in the first string.    |

**Note:** All three concatenation operators leave the second string as is. If the string contains any leading blanks, those leading blanks are always included in the resulting string.

Figure 4.10 show some examples.

```

DCL &first      *CHAR  10
DCL &last       *CHAR  10
DCL &full       *CHAR  25

CHGVAR &first 'JOHN'
CHGVAR &last  'DOE'

CHGVAR &full (&first *CAT &last)
          /* &FULL contains 'JOHN      DOE          ' */

CHGVAR &full (&first *BCAT &last)
          /* &FULL contains 'JOHN DOE          ' */

CHGVAR &full (&first *TCAT &last)
          /* &FULL contains 'JOHNDOE          ' */

```

Figure 4.10: Examples of concatenation and the results.

---

**Note:** You can use symbols ||, |> and |< respectively instead of \*CAT, \*BCAT, and \*TCAT. However, using \*CAT, \*BCAT, and \*TCAT seems the wiser choice because they are more explicit (you don't need to memorize the meaning of symbols).

Also, no matter what printer you have, they are always printable. Some printers might have a problem with the | character (and even with > and <) because they might be replaced by other characters or even by blank spaces (which makes your CL source listing unreadable).

Also, consider that if you download CL source code to a PC, the | character might be replaced by something totally unexpected.

---

As explained previously, more than two decimal variables can be combined with arithmetic operators. The same applies to the concatenation operators and the %SST function, which can be combined in many ways.

## SIMULATING ARRAYS IN CL

CL does not support arrays. If you need an array in a CL procedure, you can simulate one by using a very long character string and the %SST function.

Suppose you need an array of 50 object names (each one being 10 characters long). You could declare a 500-character variable as shown in Figure 4.11.

```
DCL &obj_arr *CHAR 500
```

*Figure 4.11: Defining a variable to be used as an array.*

Now you can reference the first element of this pseudo-array by using %SST(&obj\_arr 1 10). The second element is %SST(&obj\_arr 11 10), and so on, all the way to the 50th element—which would be %SST(&obj\_arr &n 10),

If you need to reference the  $n$ th element (that is, where the element number is a variable), you can code something like `%SST(&obj_arr &n 10)`, where `&N` has been declared as a decimal variable, and contains the number of the byte where the element should begin. Figure 4.12 shows two examples.

```
/* Changing the second element */
CHGVAR %SST(&obj_arr 11 10) 'XYZ'

/* Assigning the second element to another variable */
CHGVAR &x %SST(&obj_arr 11 10)
```

Figure 4.12: Examples of referencing portions of a variable like an array.

## BINARY CONVERSION

The `%BIN` (or `%BINARY`) function extracts binary values from character variables and copies binary values to character variables. The portion of the character variable must be either 2 or 4 bytes, depending on the size of the binary number being received. The `%BIN` function is very similar to `%SST`. Its use is illustrated in Figure 4.13.

```
PGM (&counter)

DCL &counter *CHAR 2
DCL &numeric *DEC 5

CHGVAR &numeric %BIN(&counter 1 2)
```

Figure 4.13: Using `%BIN` to convert binary data to numeric

Using `%BIN(&COUNTER 1 2)` literally means to extract the first two bytes of `&COUNTER` (as `%SST` would have done), and to interpret them as a numeric value. Therefore, the result can be assigned to `&NUMERIC`, which is a decimal variable.

In the example shown in Figure 4.13, `&COUNTER` is exactly two characters long. Therefore, it feels strange to “extract its first two bytes.” In this case, `%BIN` can be abbreviated to eliminate the second and third parameters (leaving the code shown in Figure 4.14).

```
CHGVAR &numeric %BIN(&counter)
```

*Figure 4.14: An example of an abbreviated form of the %BIN keyword.*

In general, you can omit the second and third parameters of `%BIN` if you want to use the full length of the character variable on which `%BIN` is to operate (`&COUNTER`, in this case).

This function can be used to advantage in CL procedures that act as command processing programs (CPPs), which receive lists or varying length parameters from the command processor. The command processor passes two-byte prefixes in these cases, which need to be interpreted within the CL procedure.

You can use `%BIN` as a pseudo-variable in the `VAR`, rather than `VALUE`, parameter of `CHGVAR` (see Figure 4.15).

```
DCL &counter      *DEC      5
DCL &count_chr   *CHAR      2

CHGVAR %BIN(&count_chr 1 2) &counter
CALL rpgpgm (&count_chr)
```

*Figure 4.15: An example of using %BIN in the VAR parameter.*

Because the `%BIN` function is in the `VAR` parameter, the numeric value in `&COUNTER` is first converted to character form and then assigned to the first two characters of `&COUNT_CHR`. Again, because the receiving variable is only two characters long, you can omit the second and third parameters of `%BIN`. Figure 4.16 shows an example.

```
CHGVAR %BIN(&count_chr) &counter
```

Figure 4.16: Coding %BIN without the second and third parameters.

Besides these uses, %BIN can be used in the COND parameter of an IF statement or in any command parameter that expects a numeric value. The statements shown in Figure 4.17 are both valid.

```
IF (&counter *EQ %BIN(&alpha 1 4))  
  CRTPF FILE(qtemp/test) RCDLEN(%BIN(&size))
```

Figure 4.17: Other examples of %BIN usage.

## Logical Operations

Logical variables also can be manipulated using the \*NOT, \*AND, and \*OR operators. These operations can be represented by the symbols  $\neg$ , &, and |. You can use either method, but remember that some printers might have problems with the first and last symbol. Also, downloading the code to a PC could turn those symbols into something quite different.

- \*NOT changes a single logical variable from TRUE to FALSE or vice versa.
- \*AND combines two logical variables into a single logical value. If both variables are TRUE, the result is TRUE. Any other combination yields a FALSE result.
- \*OR combines two logical variables into a single logical value. If either variable is TRUE, the result is TRUE. If both variables are FALSE, the result is FALSE.

Figure 4.18 shows some examples of the use of logical operators.

```
DCL &true      *LGL      1 VALUE('1')
DCL &false     *LGL      1 VALUE('0')
DCL &result    *LGL      1

CHGVAR &result (*NOT &true)
/* &RESULT is FALSE */
CHGVAR &result (&true *AND &false)
/* &RESULT is FALSE */

CHGVAR &result (&true *OR &false)
/* &RESULT is TRUE */
```

Figure 4.18: Examples of logical operations..

## EXPRESSIONS AND OPERATOR HIERARCHY

An *expression* is a combination of two or more variables or constants that use operators and can be evaluated into a single value. Like variables, expressions can be of character, decimal, or logical type.

When coding expressions, be careful not to combine values of a type not supported by an operator. For example, the \*CAT operator should be used only on two character values and never on numeric or logical values.

Among the arithmetic operators, multiplication and division (\* and /) take precedence over addition and subtraction (+ and -). An expression such as  $2+3*4$  yields 14, not 20, because the multiplication (3 times 4) is evaluated first, which gives an intermediate expression of  $2+12$ .

If this natural order of evaluation has to be overridden, you must use parentheses to group the values that should be combined first. Therefore, the expression  $(2+3)*4$  yields 20, not 14, because the parentheses force the system to evaluate  $2+3$  first, and then multiply the result by 4.

Because they have the same hierarchy, all character operators (\*CAT, \*BCAT, and \*TCAT) are evaluated from left to right without exception.

Among the logical operators, \*NOT is evaluated first, followed by \*AND, and then \*OR. Again, you can circumvent this natural order by using parentheses.

## THE CVTDAT COMMAND

Out of the hundreds of commands available, Convert Date (CVTDAT) stands out because it performs an often-needed function. With it, you can convert a date value from any format to any other (with the option of placing date-separator characters in the result or omitting them). CVTDAT has five parameters:

- **DATE:** The date being converted. This must be a character value (either variable or constant).
- **TOVAR:** Name of the CL variable that will receive the converted date. This must be a character variable that is from 5 to 10 bytes long, depending on the format desired.
- **FROMFMT:** Indicates what format DATE is in. You can enter any of the values listed in Table 4.3.
- **TOFMT:** The format you want TOVAR to be in. TOFMT accepts the same values as FROMFMT (listed in Table 4.3).
- **TOSEP:** The date separator character that is desired for TOVAR. Valid values are listed in Table 4.4.

**Table 4.3: Valid Formats for the FROMFMT Parameter**

| <b>Valid Formats</b> | <b>Description</b>   |
|----------------------|--|
| *SYSVAL              | Use if DATE is in the default date format indicated by system value QDATFMT.   |
| *JOB                 | Use if DATE is in the default date format specified for this job. The job's date format defaults to system value QDATFMT, but can be changed with the Change Job (CHGJOB) command. |
| *MDY                 | Use if DATE is in the format MMDDYY.   |
| *MDYY                | Use if DATE is in the format MMDDYYYY.   |
| *YMD                 | Use if DATE is in the format YYMMDD.   |

**Table 4.3: Valid Formats for the FROMFMT Parameter, continued**

| <b>Valid Formats</b> | <b>Description</b>  |
|----------------------|---|
| *YYMD                | Use if DATE is in the format YYYYMMDD.  |
| *DMY                 | Use if DATE is in the format DDMMYY.  |
| *DMYY                | Use if DATE is in the format DDMMYYYY   |
| *CYMD                | Use if DATE is in the format CYYMMDD.   |
| *JUL                 | Use if DATE is in the format (YYYYDDD).                                       |
| *LONGJUL             | Use if DATE is in the format (YYYYDDD).                                       |
| *ISO                 | Use if DATE is in the International Standards Organization format YYYY-MM-DD. |
| *JIS                 | Use if DATE is in the Japanese Industrial Standards format YYYY-MM-DD.        |
| *USA                 | Use if DATE is in the United States format MM/DD/YYYY.                        |
| *EUR                 | Use if DATE is in the European format DD.MM.YYYY.                             |

**Table 4.4: Valid Date Separator Characters for the TOSEP Parameter**

| <b>Valid Formats</b> | <b>Description</b>  |
|----------------------|---|
| *NONE                | Use if no separator characters are desired.   |
| *SYSVAL              | Use if TOVAR is to be formatted with the default date separator character, as retrieved from system value QDATSEP.  |
| *JOB                 | Use if TOVAR is to be formatted with the default date separator character for the job. The job's date separator character defaults to system value QDATFMT, but can be changed with CHGJOB. |
| *BLANK               | Use blanks as separators.   |

**Table 4.4: Valid Date Separator Characters for the TOSEP Parameter, continued**

| Valid Formats | Description                |
|---------------|----------------------------|
| '/'           | Use slashes as separators. |
| '-'           | Use dashes as separators.  |
| '.'           | Use periods as separators. |
| ','           | Use commas as separators.  |

Figure 4.19 shows an example of the CVTDAT command.

```

DCL &filedate *CHAR 6
DCL &dspdate *CHAR 6

CVTDAT DATE(&filedate) TOVAR(&dspdate) +
        FROMFMT(*YMD) TOFMT(*MDY) TOSEP('/')
```

Figure 4.19: An example of the CVTDAT command.

A file contains a date in the YYMMDD format. This date is stored in variable &FILEDATE. The same date must be shown on the display in the MMDDYY format using slashes as separator characters.

When you use CVTDAT, keep in mind that all fields must have valid date values. For instance, you cannot convert a date of 02/29/99 to some other format.

Date formats that express the year as a two-digit number (\*MDY, \*DMY, \*YDM, \*JUL) can contain dates between January 1, 1940 and December 31, 2039. Date formats that express the year as a four-digit number can contain dates between August 24, 1928 and May 9, 2071.