You will learn how to process data in ABAP…

- Assigning Values
- Resetting Values to Initial Values
- Numerical Operations
- Processing Character Strings
- Specifying Offset Values for Data Objects
- Type Conversion

In ABAP/4, you can assign values to data objects in both declarative and operational statements. In declarative statements, you assign start values to data objects on declaration. To do this, you use the following:

Syntax:

```
<declarative keyword> [TYPE] <f> VALUE  <initial value>.
```

<declarative keyword> is a place holder for DATA, COCTANTS, or STATICS. The value <initial value> is assigned to the data object <f>.

Example:

```
DATA   NAME   TYPE   C(10)   VALUE   'SATYAM'.
```

This statement initializes the data object NAME with SATYAM.

- Basic Assignment Operations
- Assigning values with Offset specifications
- Copying values between components of Field Strings

Basic Assignment Operations

Syntax: MOVE <f1> TO <f2>

The MOVE statement transports the contents of the source field <f1>, which can be any data object, to the target field <f2>

The MOVE statement and the assignment operator have the same function. The above statement when written with assignment operator (=) has the form:

f2 = f1.

Example:

```
DATA: T(10),
        NUMBER TYPE P DECIMALS 1,
        COUNT TYPE P DECIMALS 1.
T  =  1111.
MOVE  ‘ 5.3 ‘  TO  NUMBER.
COUNT  =  NUMBER.
```

The result of this assignment is that fields T, NUMBER, and COUNT contain the values ‘ 1111 ’, ‘ 5.3 ‘, and ‘ 5.3 ‘.

Assigning values with Offset specifications

You can Specify offsets and lengths for elementary data objects in every ABAP/4 statement.

Syntax:

```
MOVE <f1> [+<p1>][(<l1>)] TO <f2> [+<p2>][(<l2>)].
```

The contents of the section of field <f1>, which begins at the position <p1>+1 and has a length of <l1>, are assigned to field <f2> where they overwrite the section which begins at the position <p2>+1 and has a length of <l2>.

In the MOVE statement, all offset and length specifications can be variables.

Example:

```
DATA:             F1(8) VALUE  'ABCDEFGH',
                  F2(8).
DATA:             O TYPE I VALUE 2,
                  L TYPE I VALUE 4.
MOVE  F1  TO  F2.                     Write / F2.
MOVE  F1+O(L)  TO  F2.                Write / F2.
MOVE  F1  TO  F2 +O(L).               Write / F2.
CLEAR F2.
MOVE  F1  TO  F2 +O(L).               Write / F2.
MOVE  F1+O(L)  TO  F2+O(L).           Write / F2.
```

The above example produces the following output:

```
ABCDEFGH
CDEF
CDABCD
CDEF
```

Copying values between components of Field Strings

Syntax:

```
MOVE-CORRESPONDING <STRING1>  TO <STRING2>.
```

This statement assigns the contents of the components of field string <string1> to those components of field string <string2> which have the same names.

The system executes a MOVE statement for each name pair as follows:

```
MOVE STRING1- <component> TO STRING2- <component>
```

Example:

```
DATA:      BEGIN OF ADDRESS,
                     FIRSTNAME(20) VALUE ‘FRED’,
                     SURNAME(20)    VALUE ‘Flintstone’,
                     INITIALS(4)       VALUE ‘FF’,
                     STREET(20)       VALUE ‘Cave Avenue’,
                     NUMBER TYPE I VALUE ‘11’,
                     POSTCODE TYPE N VALUE ‘98765’,
                     CITY(20)       VALUE ‘Bedrock’.
           END OF ADDRESS.
DATA:      BEGIN OF NAME,
                     FIRSTNAME(20),       SURNAME(20),
                     INITIALS(4),
                     TITLE(10)  VALUE ‘Mister’,
           END OF NAME.
      MOVE-CORRESPONDING ADDRESS TO NAME.
```

In the example , the values of NAME-SURAME, NAME-FIRSTNAME and NAME-INITIALS are set to ‘Flintstone’, ‘Fred’, and ‘FF’. NAME-TITLE retains the value ‘Mister’.

- **Basic form of the WRITE TO statement**
- **Specifying the source field at runtime**
- **Writing values with Offset specifications**

Basic form of the WRITE TO statement

Syntax: WRITE <f1> TO <f2> [<option>].

This statement converts the contents of a data object <f1> to type C, and places the string in the variable <f2>. The data type of <f1> must be convertible into a character field; if it is not, a syntax or runtime error occurs.

When assigning values to data objects with WRITE TO statement you can use all the formatting options available except UNDER and NO-GAP.

Formatting options for all data types:

Options	Function
LEFT- JUSTIFIED	Output is left-justified.
CENTERED	Output is centered.
RIGHT- JUSTIFIED	Output is right-justified.
UNDER <g>	Output starts directly under field <g>.
NO-GAP	The blank after field <f> is omitted.
USING EDIT MASK <m>	Specifies format template <m>.
USING NO EDIT MASK	Deactivates a format template specified in the ABAP Dictionary.
NO-ZERO	If a field contains only zeros, these are replaced by blanks. For type C and N fields, leading zeros are replaced automatically.

Formatting options for all data types:

Options	Function
NO-SIGN	The leading sign is not displayed on the screen.
DECIMALS <d>	<d> defines the number of digits after the decimal point.
EXPONENT <e>	In type F fields, the exponent is defined in <e>.
ROUND <r>	Type P fields are multiplied by 10**(-r) and then rounded.
CURRENCY <c>	Format according to currency <c> in table TCURX.
UNIT <u>	The number of decimal places is fixed according to unit <u> specified in table T006 for type P fields.

Formatting options for all data types:

Options	Function
DD/MM/YY	Separators as defined in user’s master record.
MM/DD/YY	Separators as defined in user’s master record.
DD/MM/YYYY	Separators as defined in user’s master record.
MM/DD/YYYY	Separators as defined in user’s master record.
DDMMYY	No separators.
MMDDYY	No separators.
YYMMDD	No separators.

Example:

```
DATA: NUMBER TYPE F VALUE '4.3',

      TEXT(10),
      FLOAT TYPE F,
      PACK TYPE P DECIMALS 1.
WRITE NUMBER.

WRITE NUMBER TO TEXT EXPONENT 2.
WRITE / TEXT.

WRITE NUMBER TO FLOAT.
WRITE / FLOAT.

WRITE NUMBER TO PACK.
WRITE / PACK.

MOVE NUMBER TO PACK.
WRITE / PACK.
```

This produces the following output:

```
4.30000000000000E+00

0.043E+02

1.50454551753894E-153

20342<33452;30,3

4.3
```

Specifying the source field at runtime:

To specify the name of the source field dynamically as the contents of another field, specify the name of the field containing the name of the source field in parentheses.

Syntax:

```
WRITE (<f>) TO <g>.
```

The system places the value of the data object assigned to <f> in <g>.

Example:

```
DATA: NAME(10) VALUE 'SOURCE',
SOURCE(10) VALUE 'Antony',
TARGET(10).

...
WRITE (NAME) TO TARGET.
WRITE: TARGET.
```

This produces the output

```
Antony
```

Writing values with Offset specifications

Syntax:

```
WRITE <f1> [+<p1>][(<l1>)] TO <f2> [+<p2>][(<l2>)].
```

The contents of the section of field <f1>, which begins at the position <p1>+1 and has a length of <l1>, are assigned to field <f2> where they overwrite the section which begins at the position <p2>+1 and has a length of <l2>.

In the MOVE statement, all offset and length specifications can be variables.

Example:

```
Data:STRING(20),
     NUMBER(8)       TYPE   C      VALUE          '123456',
     OFFSET          TYPE   I      VALUE   8,
     LENGTH          TYPE   I      VALUE   12.

WRITE NUMBER+(6) TO STRING +OFFSET(LENGTH) LEFT-JUSTIFIED.
WRITE: / STRING.
CLEAR   STRING.

WRITE NUMBER+(6) TO STRING +OFFSET(LENGTH) CENTERED.
WRITE: / STRING.
CLEAR   STRING.

WRITE NUMBER+(6) TO STRING +OFFSET(LENGTH) RIGHT-JUSTIFIED.
WRITE: / STRING.
CLEAR   STRING.
```

This produces the following output:

```
123456

   123456

      123456
```

The first WRITE statement writes the first 6 positions of the field NUMBER left-justified to the last 12 positions of the field STRING. The second WRITE statement writes the first 6 positions of the field NUMBER centered to the last 12 positions of the field STRING. The third WRITE statement writes the first 6 positions of the field NUMBER right-justified to the last 12 positions of the field STRING.

To reset a variable to the appropriate initial value for its type, use the CLEAR statement.

Syntax:

```
CLEAR < f >.
```

This statement has different effects for different data types:

- **Elementary ABAP types**

  The CLEAR statement sets the value of an elementary variable to the initial value specified for its type and not to the starting value that you specified in the VALUE addition to the DATA statement.

- **References**

  The CLEAR statement resets a reference variable to its initial value, that is, so that it does not point to an object.

- **Structures**

  The CLEAR statement resets the individual components of a structure to their respective initial values.

•Internal tables

The CLEAR statement deletes the entire contents of an internal table

Example:

```
DATA NUMBER TYPE I VALUE ‘10’.

WRITE NUMBER.

CLEAR NUMBER.

WRITE / NUMBER.
```

This produces the following output:

```
10

0
```

The clear statement resets the contents of the field umber from 10 to initial value 0.

•Introduction

•Performing Arithmetic Operations

•Using Mathematical Functions

•Processing Packed Numbers

•Processing Date and Time Fields

Introduction:

To assign the result of a mathematical calculation to a variable, use the COMPUTE statement or the assignment operator (=).

Syntax :

```
COMPUTE <n> = <expression>.
```

COMPUTE is optional, so you can also write the statement as follows:

<n> = <expression>.

The result of the mathematical operation specified in <expression> is assigned to the field <n>.

NOTE: The data type of the result field influences the accuracy of the entire calculation.

ABAP interprets mathematical expressions in the following order:

- Expressions in parentheses
- Functions
- ** (powers)
- *, / , MOD, DIV (multiplication, division)
- +, - (addition, subtraction)

•Basic Arithmetic Operations

•Performing Arithmetic Operations on Field Strings

•Adding Sequences of Fields

Basic Arithmetic Operations:

ABAP supports the four basic arithmetic operations and power calculations. You can use the following arithmetic operators in mathematical expressions:

Operator	Meaning
Addition	+
Subtraction	-
Multiplication	*
Division	/
Integer division	DIV
Remainder of integer division	MOD
Powers	**

Instead of using operators in mathematical expressions, you can perform basic arithmetic operations with the keywords ADD, SUBTRACT, MULTIPLY, and DIVIDE.

The following table shows the different ways of expressing basic arithmetic operations in ABAP and their Syntax:

Operation	**Syntax - using mathematical Expression**	**Syntax - using statement keyword**
Addition	**<p> = <n> + <m>.**	**ADD <n> TO <m>.**
Subtraction	**<p> = <m> - <n>.**	**SUBTRACT <n> FROM <m>.**
Multiplication	**<p> = <m> * <n>.**	**MULTIPLY <m> BY <n>.**
Division	**<p> = <m> / <n>.**	**DIVIDE <m> BY <n>.**
Integer division	**<p> = <m> DIV <n>.**	**---**
Remainder of division	**<p> = <m> MOD <n>.**	**---**
Powers	**<p> = <m> ** <n>.**	**---**

NOTE:

•When using mathematical expressions, please note that the operators +, -, *, **, and /, as well as opening and closing parentheses, are interpreted as words in ABAP and must therefore be preceded and followed by blanks.

•When you combine arithmetic expressions, ABAP performs the calculation from left to right, with one exception: Powers are calculated from right to left.

Example1:

```
DATA: COUNTER TYPE I.
COMPUTE COUNTER = COUNTER + 1.
      COUNTER = COUNTER + 1.
ADD 1 TO COUNTER.
```

Here, the three operational statements perform the same arithmetic operation, i.e. adding 1 to the contents of the field COUNTER and assigning the result to COUNTER.

Example2:

This example shows the different types of division.

```
DATA: PACK TYPE P DECIMALS 4,
N TYPE F VALUE '+5.2',
M TYPE F VALUE '+1.1'.

PACK = N / M.
WRITE PACK.

PACK = N DIV M.
WRITE / PACK.

PACK = N MOD M.
WRITE /PACK.
```

The output appears as follows:

```
4.7273

4.0000

0.8000
```

In the same way that you can transfer values, component by component, between structures using the MOVE-CORRESPONDING statement, you can also perform arithmetic operations between the components of structures using the following statements:

- **ADD-CORRESPONDING**
- **SUBTRACT-CORRESPONDING**
- **MULTIPLY-CORRESPONDING**
- **DIVIDE-CORRESPONDING**

ABAP performs the corresponding calculation for all components that have the same name in both structures.

Example:

```
DATA: BEGIN OF RATE,
USA TYPE F VALUE '0.6667',
FRG TYPE F VALUE '1.0',
AUT TYPE F VALUE '7.0',
END OF RATE.

DATA: BEGIN OF MONEY,
USA TYPE I VALUE 100,
FRG TYPE I VALUE 200,
AUT TYPE I VALUE 300,
END OF MONEY.

MULTIPLY-CORRESPONDING MONEY BY RATE.

WRITE / MONEY-USA.
WRITE / MONEY-FRG.
WRITE / MONEY-AUT.
```

The output appears as follows:

```
67

200

2,100
```

Here, MONEY-USA is multiplied by RATE-USA and so on.

There are variants of the ADD statement that allow you to add sequences of fields in memory.

For example:

Adding sequences of fields and assigning the result to another field

Syntax: ADD <n$_1$> THEN <n $_2$> UNTIL <n $_z$> GIVING <m>.

If <n $_1$ >, <n $_2$ >, ... , <n $_z$ > is a sequence of equidistant fields of the same type and length in memory, they are summed and the result is assigned to <m>.

Adding sequences of fields and adding the result to the contents of another field

Syntax: ADD <n$_1$> THEN <n $_2$> UNTIL <n $_z$> TO <m>.

This statement is identical to the preceding one, with one difference: The sum of the fields is added to the existing contents of <m>.

Example:

```
DATA: BEGIN OF SERIES,
N1 TYPE I VALUE 10,
N2 TYPE I VALUE 20,
N3 TYPE I VALUE 30,
N4 TYPE I VALUE 40,
N5 TYPE I VALUE 50,
N6 TYPE I VALUE 60,
END OF SERIES.

DATA SUM TYPE I.

ADD SERIES-N1 THEN SERIES-N2 UNTIL SERIES-N5 GIVING SUM.
WRITE SUM.

ADD SERIES-N2 THEN SERIES-N3 UNTIL SERIES-N6 TO SUM.
WRITE / SUM.
```

Output

```
150

350
```

Here, the contents of components N1 to N5 are summed and assigned to the field SUM. Then, the contents of components N2 to N6 are summed and added to the value of SUM.

•Introduction

•Functions for Numeric Data Types

•Functions for Floating-Point Data Types

Introduction:

ABAP contains a range of built-in functions that you can use as mathematical expressions, or as part of a mathematical expression.

```
    [COMPUTE] <n> = <func>( <m> ).
```

The blanks between the parentheses and the argument <m> are obligatory. The result of calling the function <func> with the argument <m> is assigned to <n>.

COMPUTE is optional, so you can also write the statement as follows:

```
    <n> = <func>( <m> ).
```

The following built-in functions work with all three numeric data types (F, I, and P) as arguments.

Function	Result
ABS	Absolute value of argument.
SIGN	Sign of argument:
CEIL	Smallest integer value not smaller than the argument.
FLOOR	Largest integer value not larger than the argument.
TRUNC	Integer part of argument.
FRAC	Fraction part of argument.

$$\mathrm{SIGN}(X) = \begin{cases} 1 & \text{if} \quad X > 0 \\ 0 & \text{if} \quad X = 0 \\ -1 & \text{if} \quad X < 0 \end{cases}$$

The argument of these functions does not have to be a numeric data type. If you choose another type, it is converted to a numeric type. These functions do not change the numerical precision of a numerical operation.

Example:

```
DATA N TYPE P DECIMALS 2.
DATA M TYPE P DECIMALS 2 VALUE '-5.55'.

N = ABS( M ). WRITE: 'ABS: ', N.
N = SIGN( M ). WRITE: / 'SIGN: ', N.
N = CEIL( M ). WRITE: / 'CEIL: ', N.
N = FLOOR( M ). WRITE: / 'FLOOR:', N.
N = TRUNC( M ). WRITE: / 'TRUNC:', N.
N = FRAC( M ). WRITE: / 'FRAC: ', N.
```

The output appears as follows:

```
ABS: 5.55
SIGN: 1.00-
CEIL: 5.00-
FLOOR: 6.00-
TRUNC: 5.00-
FRAC: 0.55-
```

Exapmle2:

```
DATA: T1(10),
T2(10) VALUE '-100'.

T1 = ABS( T2 ).

WRITE T1.
```

This produces the following output:

```
100
```

Two conversions are performed. First, the contents of field T2 (type C) are converted to type P. Then the system processes the ABS function using the results of the conversion. Then, during the assignment to the type C field T1, the result of the function is converted back to type C.

The following built-in functions work with floating point numbers (data type F) as an argument.

Function	Meaning
ACOS, ASIN,	Trigonometric functions.
ATAN; COS,	
SIN, TAN	
COSH, SINH, TANH	Hyperbolic functions.
EXP	Exponential function with base e (e=2.7182818285).
LOG	Natural logarithm with base e.
LOG10	Logarithm with base 10.
SQRT	Square root.

The argument of these functions does not have to be a floating point field. If you choose another type, it is converted to type F. The functions themselves have the data type F. This can change the numerical precision of a numerical operation.

Example:

```
DATA: RESULT TYPE F,
      PI(10) VALUE '3.141592654'.

      RESULT = COS( PI ).

WRITE RESULT.
```

The output is -1.00000000000000E+00 . The character field PI is automatically converted to a type F field before the calculation is performed.

For calculations in business applications, use packed numbers.

If the program attribute Fixed point arithmetic is not set, type P fields are interpreted as integers without decimal places. The decimal places that you specify in the DECIMALS addition of the TYPES or DATA statement only affect how the field is formatted in the WRITE statement.

Example1:

```
DATA: PACK TYPE P DECIMALS 2.
PACK = '12345'.
WRITE PACK.
```

If the program attribute Fixed point arithmetic is not set, the output is as follows:

```
123.45
```

If the program attribute *Fixed point arithmetic* is set, the output is as follows:

```
12,345.00
```

If the Fixed point arithmetic attribute is set, the decimal places are also taken into account in arithmetic operations. In calculations with packed numbers, in ABAP, intermediate results are calculated using up to 31 digits (before and after the decimal point). You should therefore always set the Fixed point arithmetic attribute when you use type P fields.

Example2:

```
DATA: PACK TYPE P.
PACK = 1 / 3 * 3.
WRITE PACK.
```

If you have not set the Fixed point arithmetic attribute, the result is 0, since the calculation is performed using integer accuracy, and the result is therefore rounded internally to 0.

If the program attribute Fixed point arithmetic is set, the result is 1 because the result of the division is stored internally as 0.333333333333333333333333333333 with an accuracy of up to 31 digits.

Although Date and time fields have character types, you can use date and time fields in numeric operations.

Example of a date calculation:

```
DATA: ULTIMO TYPE D.
ULTIMO = SY-DATUM.
ULTIMO+6(2) = '01'.          " = first day of this month
ULTIMO = ULTIMO - 1. " = last day of last month
```

Here, the last day of the previous month is assigned to the date field ULTIMO.

1. ULTIMO is filled with the present date.

2. Using an offset specification, the day is changed to the first day of the current month.

3. 1 is subtracted from ULTIMO. Its contents are changed to the last day of the previous month. Before performing the subtraction, the system converts ULTIMO to the number of days since 01.01.0001 and converts the result back to a date.

Example of a time calculation:

```
DATA: DIFF TYPE I,
SECONDS TYPE I,
HOURS TYPE I.

DATA: T1 TYPE T VALUE '200000',
T2 TYPE T VALUE '020000'.

DIFF = T2 - T1.
SECONDS = DIFF MOD 86400.
HOURS = SECONDS / 3600.
```

The number of hours between 02:00:00 and 20:00:00 is calculated.

1. First, the difference between the time fields is calculated. This is -64800, since T1 and T2 are converted internally into 72000 and 7200 respectively.

2. Second, with the operation MOD, this negative difference is converted to the total number of seconds. A positive difference would be unaffected by this calculation.

3. Third, the number of hours is calculated by dividing the number of seconds by 3600.

Inverted Dates:

In some cases (for example, when sorting dates in descending order), it is useful to convert a date from format D to an inverted date by using the keyword CONVERT.

Syntax:

```
CONVERT DATE <d1> INTO INVERTED-DATE <d2>.
```

Afterwards, you can convert the inverted data back into a normal date using the statement

```
CONVERT INVERTED-DATE <d1> INTO DATE <d2>.
```

These statements convert the field <d1> from the formats DATE or INVERTED-DATE to the formats INVERTED-DATE or DATE and assign it to the field <d2>.

For the conversion, ABAP forms the nine's complement.

Example:

```
DATA: ODATE TYPE D VALUE '19955011',
      IDATE LIKE ODATE.

DATA FIELD(8).

FIELD = ODATE.   WRITE / FIELD.

CONVERT DATE ODATE INTO INVERTED-DATE IDATE.

FIELD = IDATE.   WRITE / FIELD.

CONVERT INVERTED-DATE IDATE INTO DATE ODATE.

FIELD = ODATE.   WRITE / FIELD.
```

This produces the following output:

```
80049488
19955011
19955011
```

•Shifting Field Contents

•Replacing Field Contents

•Converting to Upper or Lower Case or Replacing Characters

•Converting into a Sortable Format

•Overlaying Character Fields

•Finding Character Strings

•Finding the Length of a Character String

•Condensing Field Contents

•Concatenating Character Strings

•Splitting Character Strings

•Assigning Parts of Character Strings

You can shift the contents of a field using the following variants of the SHIFT statement. SHIFT moves field contents character by character.

•Shifting a String by a Given Number of Positions

•Shifting a Structure up to a Given String

•Shifting a Structure According to the First or Last Character

Shifting a String by a Given Number of Positions:

Syntax:

SHIFT <c> [BY <n> PLACES] [<mode>].

This statement shifts the field <c> by <n> positions. If you omit BY <n> PLACES, <n> is interpreted as one. If <n> is 0 or negative, <c> remains unchanged. If <n> exceeds the length of <c>, <c> is filled out with blanks. <n> can be variable.

With the different <mode> options, you can shift the field <c> in the following ways:

- **<mode> is LEFT:**

  Shifts the field contents <n> places to the left and adds <n> blanks at the right-hand end of the field (default).

- **<mode> is RIGHT:**

  Shift <n> positions to the right and adds <n> blanks at the left-hand end of the field.

- **<mode> is CIRCULAR:**

  Shift <n> positions to the left so that <n> characters on the left appear on the right.

Example:

```
DATA: T(10) VALUE 'abcdefghij',
STRING LIKE T.

STRING = T.
WRITE STRING.

SHIFT STRING.
WRITE / STRING.

STRING = T.
SHIFT STRING BY 3 PLACES LEFT.
WRITE / STRING.

STRING = T.
SHIFT STRING BY 3 PLACES RIGHT.
WRITE / STRING.

STRING = T.
SHIFT STRING BY 3 PLACES CIRCULAR.
WRITE / STRING.
```

This produces the following output:

```
Abcdefghij
bcdefghij
defghij

   abcdefg
defghijabc
```

Syntax:

SHIFT <c> UP TO <str> <mode>.

This statement searches the field contents of <c> until it finds the string <str> and shifts the field <c> up to the edge of the field. The <mode> options are the same as described above. <str> can be a variable.

If <str> is not found in <c>, SY-SUBRC is set to 4 and <c> is not shifted. Otherwise, SY-SUBRC is set to 0.

Example:

```
DATA: T(10) VALUE 'abcdefghij',
STRING LIKE T,
STR(2) VALUE 'ef'.
STRING = T.
WRITE STRING.
SHIFT STRING UP TO STR.
WRITE / STRING.
STRING = T.
SHIFT STRING UP TO STR LEFT.
WRITE / STRING.
STRING = T.
SHIFT STRING UP TO STR RIGHT.
WRITE / STRING.
STRING = T.
SHIFT STRING UP TO STR CIRCULAR.
WRITE / STRING.
```

This produces the following output:

```
abcdefghij

efghij

efghij

    abcdef

efghijabcd
```

Syntax:

```
SHIFT <c> LEFT DELETING LEADING <str>.

SHIFT <c> RIGHT DELETING TRAILING <str>.
```

This statement shifts the field <c> to the left or to the right, provided the first character on the left or the last character on the right occur in <str>. The right or left of the field is then padded with blanks. <str> can be a variable.

Example:

```
DATA: T(14) VALUE ' abcdefghij',
STRING LIKE T,
STR(6) VALUE 'ghijkl'.

STRING = T.
WRITE STRING.

SHIFT STRING LEFT DELETING LEADING SPACE.
WRITE / STRING.

STRING = T.
SHIFT STRING RIGHT DELETING TRAILING STR.
WRITE / STRING.
```

Output:

```
   abcdefghij
abcdefghij
       abcdef
```

Syntax:

REPLACE <str1> WITH <str2> INTO <c> [LENGTH <l>].

This statement searches the field <c> for the first occurrence of the first <l> positions of the pattern <str1>. If no length is specified, it searches for the pattern <str1> in its full length.

Then, the statement replaces the first occurrence of the pattern <str1> in field <c> with the string <str2>. If a length <l> was specified, only the relevant part of the pattern is replaced.

If the return code value of the system field SY-SUBRC is set to 0, this indicates that <str1> was found in <c> and replaced by <str2>. A return code value other than 0 means that nothing was replaced.

<str1>, <str2>, and <len> can be variables.

Example:

```
DATA: T(10) VALUE 'abcdefghij',
STRING LIKE T,
STR1(4) VALUE 'cdef',
STR2(4) VALUE 'klmn',
STR3(2) VALUE 'kl',
STR4(6) VALUE 'klmnop',
LEN TYPE I VALUE 2.

STRING = T.
WRITE STRING.

REPLACE STR1 WITH STR2 INTO STRING.
WRITE / STRING.

STRING = T.
REPLACE STR1 WITH STR2 INTO STRING LENGTH LEN.
WRITE / STRING.

STRING = T.
REPLACE STR1 WITH STR3 INTO STRING.
WRITE / STRING.

STRING = T.
REPLACE STR1 WITH STR4 INTO STRING.
WRITE / STRING.
```

The output appears as follows:

```
abcdefghij
abklmnghij
abklmnefgh
abklghij
abklmnopgh
```

Note how, in the last line, the field STRING is truncated on the right. The search pattern 'cdef' of length 4 is replaced by 'klmnop' of length 6. Then, the rest of the field STRING is filled up to the end of the field.

The TRANSLATE statement converts characters into upper or lower case, or uses substitution rules to convert all occurrences of one character to another character.

•Converting to Upper or Lower Case

•Replacing Characters

Converting to Upper or Lower Case:

Syntax:

TRANSLATE <c> TO UPPER CASE.

TRANSLATE <c> TO LOWER CASE.

These statements convert all lower case letters in the field <c> to upper case or lower case respectively.

Replacing Characters:

Syntax:

```
TRANSLATE <c> USING <r>.
```

This statement replaces all characters in field <c> according to the substitution rule stored in field <r> . <r> contains pairs of letters, where the first letter of each pair is replaced by the second letter. <r> can be a variable.

Example:

```
DATA: T(10) VALUE 'AbCdEfGhIj',
STRING LIKE T,
RULE(20) VALUE 'AxbXCydYEzfZ'.

STRING = T.
WRITE STRING.

TRANSLATE STRING TO UPPER CASE.
WRITE / STRING.

STRING = T.
TRANSLATE STRING TO LOWER CASE.
WRITE / STRING.

STRING = T.
TRANSLATE STRING USING RULE.
WRITE / STRING.
```

The output appears as follows:

```
AbCdEfGhIj

ABCDEFGHIJ

abcdefghij

xXyYzZGhIj
```

The CONVERT TEXT statement converts strings into a format that can be sorted alphabetically.

Syntax:

```
CONVERT TEXT <c> INTO SORTABLE CODE <sc>.
```

This statement writes a string <c> to a sortable target field <sc>. The field <c> must be of type C and the field <sc> must be of type X with a minimum size of 16 times the size of <c>.

These sorts are applied to internal tables and extract datasets. If you sort unconverted character fields, the system creates an order that corresponds to the platform-specific internal coding of the individual letters. The conversion CONVERT TEXT creates target fields in such a way that, after sorting the target fields, the order of the corresponding character fields is alphabetical.

The OVERLAY statement overlays one string with another:

Syntax:

```
OVERLAY <c1> WITH <c2> [ONLY <str>].
```

This statement overlays all positions in field <c1> containing letters which occur in <str> with the contents of <c2>. <c2> remains unchanged. If you omit ONLY <str>, all positions of <c1> containing spaces are overwritten.

If at least one character in <c1> was replaced , SY-SUBRC is set to 0 . In all other cases, SY-SUBRC is set to 4. If <c1> is longer than <c2>, it is overlaid only in the length of <c2>.

Example:

```
DATA: T(10) VALUE 'a c e g i ',
STRING LIKE T,
OVER(10) VALUE 'ABCDEFGHIJ',
STR(2) VALUE 'ai'.

STRING = T.
WRITE STRING.
WRITE / OVER.

OVERLAY STRING WITH OVER.
WRITE / STRING.

STRING = T.
OVERLAY STRING WITH OVER ONLY STR.
WRITE / STRING.
```

The output appears as follows:

```
a c e g i

ABCDEFGHIJ

aBcDeFgHiJ

A c e g I
```

To search a character field for a particular pattern, use the SEARCH statement as follows:

Syntax:

SEARCH <c> FOR <str> <options>.

The statement searches the field <c> for <str> starting at position <n1>.

The search string <str> can have one of the following forms.

<str>	Function
<pattern>	Searches for <pattern> (any sequence of characters). Trailing blanks are ignored.
.<pattern>.	Searches for <pattern>. Trailing blanks are not ignored.
*<pattern>	A word ending with <pattern> is sought.
<pattern>*	Searches for a word starting with <pattern>.

<option> in the SEARCH FOR statement can be any of the following:

- **ABBREVIATED**

  Searches the field <c> for a word containing the string in <str>. The characters can be separated by other characters. The first letter of the word and the string <str> must be the same.

- **STARTING AT <n1>**

  Searches the field <c> for <str> starting at position <n1>. The result SY-FDPOS refers to the offset relative to <n1> and not to the start of the field.

- **ENDING AT <n2>**

  Searches the field <c> for <str> up to position <n2>.

- **AND MARK**

  If the search string is found, all the characters in the search string (and all the characters in between when using ABBREVIATED) are converted to upper case.

Example:

```
DATA STRING(30) VALUE 'This is a little sentence.'.

WRITE: / 'Searched', 'SY-SUBRC', 'SY-FDPOS'.
ULINE /1(26).

SEARCH STRING FOR 'X'.
WRITE: / 'X', SY-SUBRC UNDER 'SY-SUBRC',
          SY-FDPOS UNDER 'SY-FDPOS'

SEARCH STRING FOR 'itt '.
WRITE: / 'itt   ', SY-SUBRC UNDER 'SY-SUBRC',
            SY-FDPOS UNDER 'SY-FDPOS'

SEARCH STRING FOR '.e .'.
WRITE: / '.e .', SY-SUBRC UNDER 'SY-SUBRC',
SY-FDPOS UNDER 'SY-FDPOS'.

SEARCH STRING FOR '*e'.
WRITE: / '*e ', SY-SUBRC UNDER 'SY-SUBRC',
SY-FDPOS UNDER 'SY-FDPOS'.

SEARCH STRING FOR 's*'.
WRITE: / 's* ', SY-SUBRC UNDER 'SY-SUBRC',
SY-FDPOS UNDER 'SY-FDPOS'.
```

The output appears as follows:

```
SEARCHED SY-SUBRC SY-FDPOS
X              4        0
itt          0       11
.e .          0       15
*e             0       10
s*             0       17
```

Example:

```
DATA: STRING(30) VALUE 'This is a fast first example.',
POS TYPE I,
OFF TYPE I.
WRITE / STRING.
SEARCH STRING FOR 'ft' ABBREVIATED.
WRITE: / 'SY-FDPOS:', SY-FDPOS.
POS = SY-FDPOS + 2.
SEARCH STRING FOR 'ft' ABBREVIATED STARTING AT POS AND MARK.
WRITE / STRING.
WRITE: / 'SY-FDPOS:', SY-FDPOS.
OFF = POS + SY-FDPOS -1.
WRITE: / 'Off:', OFF.
```

The output appears as follows:

```
This is a fast first example.

SY-FDPOS:    10

This is a fast FIRST example.

SY-FDPOS:    4

Off:       15
```

Note that in order to find the second word containing 'ft' after finding the word 'fast', you have to add 2 to the offset SY-FDPOS and start the search at the position POS. Otherwise, the word 'fast' would be found again. To obtain the offset of 'first' in relation to the start of the field STRING, it is calculated from POS and SY-FDPOS.

The ABAP function STRLEN returns the length of a string up to the last character that is not a space.

Syntax:

```
[COMPUTE] <n> = STRLEN( <c> ).
```

STRLEN processes any operand <c> as a character data type, regardless of its real type. There is no type conversion.

Example:

```
DATA: INT TYPE I,
WORD1(20) VALUE '12345'.
   WORD2(20).
   WORD3(20) VALUE ' 4 '.

INT = STRLEN( WORD1 ). WRITE INT.

INT = STRLEN( WORD2 ). WRITE / INT.

INT = STRLEN( WORD3 ). WRITE / INT.
```

The results are 5 , 0, and 4 respectively.

The CONDENSE statement deletes redundant spaces from a string:

Syntax:

```
CONDENSE <c> [NO-GAPS].
```

This statement removes any leading blanks in the field <c> and replaces other sequences of blanks by exactly one blank. The result is a left-justified sequence of words, each separated by one blank. If the addition NO-GAPS is specified, all blanks are removed.

Example:

```
DATA: STRING(25) VALUE ' one two three four',
LEN TYPE I.
LEN = STRLEN( STRING ).
WRITE: STRING, '!'.
WRITE: / 'Length: ', LEN.
CONDENSE STRING.
LEN = STRLEN( STRING ).
WRITE: STRING, '!'.
WRITE: / 'Length: ', LEN.
CONDENSE STRING NO-GAPS.
LEN = STRLEN( STRING ).
WRITE: STRING, '!'.
WRITE: / 'Length: ', LEN.
```

The output appears as follows:

```
one two three four !
Length:          25
one two three four !
Length:          18
onetwothreefour !
Length:          15
```

Note that the total length of the field STRING remains unchanged, but that the deleted blanks appear again on the right.

The CONCATENATE statement combines two or more separate strings into one.

Syntax:

> **CONCATENATE <c1> ... <cn> INTO <c> [SEPARATED BY <s>].**

> **This statement concatenates the character fields <c1> to <cn> and assigns the result to <c>. The system ignores spaces at the end of the individual source strings.**
>
> **The addition SEPARATED BY <s> allows you to specify a character field <s> which is placed in its defined length between the individual fields.**

Example:

```
DATA: C1(10) VALUE 'Sum',
C2(3) VALUE 'mer',
C3(5) VALUE 'holi ',
C4(10) VALUE 'day',
C5(30),
SEP(3) VALUE ' - '.

CONCATENATE C1 C2 C3 C4 INTO C5.
WRITE C5.

CONCATENATE C1 C2 C3 C4 INTO C5 SEPARATED BY SEP.

WRITE / C5.
```

The output appears as follows:

```
Summerholiday
Sum - mer - holi – day
```

In C1 to C5, the trailing blanks are ignored. The separator SEP retains them.

To split a character string into two or more smaller strings, use the SPLIT statement as follows:

Syntax:

> **SPLIT <c> AT <del> INTO <c1> ... <cn>.**
>
> **The system searches the field <c> for the separator <del>. The parts before and after the separator are placed in the target fields <c1> ... <cn>.**
>
> **If you do not specify enough target fields, the last target field is filled with the rest of the field <c> and still contains delimiters.**
>
> **You can also split a string into the individual lines of an internal table as follows:**

Syntax:

> **SPLIT <c> AT <del> INTO TABLE <itab>.**
>
> **The system adds a new line to the internal table <itab> for each part of the string.**

Example:

```
DATA: STRING(60),
P1(20) VALUE '++++++++++++++++++++',
P2(20) VALUE '++++++++++++++++++++',
P3(20) VALUE '++++++++++++++++++++',
P4(20) VALUE '++++++++++++++++++++',
DEL(3) VALUE '***'.

STRING = ' Part 1 *** Part 2 *** Part 3 *** Part 4 *** Part 5'.
WRITE STRING.

SPLIT STRING AT DEL INTO P1 P2 P3 P4.

WRITE / P1.
WRITE / P2.
WRITE / P3.
WRITE / P4.
```

The output appears as follows:

```
Part 1 *** Part 2 *** Part 3 *** Part 4 *** Part 5
Part 1
Part 2
Part 3
Part 4 *** Part 5
```

Note that the contents of the fields P1 ...P4 are totally overwritten and that they are filled out with trailing blanks.

The following variant of the MOVE statement works only with type C fields:

Syntax:

```
MOVE <c1> TO <c2> PERCENTAGE <p> [RIGHT].
```

Copies the percentage <p> percent of the character field <c1> left-justified (or right-justified if specified with the RIGHT option) to <c2>.

The value of <p> can be a number between 0 and 100. The length to be copied from <f1> is rounded up or down to the next whole number.

If one of the arguments in the statement is not type C, the parameter PERCENTAGE is ignored.

Example:

```
DATA C1(10) VALUE 'ABCDEFGHIJ',
C2(10).

MOVE C1 TO C2 PERCENTAGE 40.

WRITE C2.

MOVE C1 TO C2 PERCENTAGE 40 RIGHT.

WRITE / C2.
```

The output appears as follows:

```
'ABCD      '
      ABCD
```

You can address a section of a string in any statement in which non-numeric elementary ABAP types or structures that do not contain internal tables occur using the following syntax:

Syntax:

<f>[+<o>][(<l>)]

By specifying an offset +<o> and a length (<l>) directly after the field name <f>, you can address the part of the field starting at position <o>+1 with length <l> as though it were an independent data object.

NOTE:

You cannot use offset and length to address a literal, a text symbol or a numeric data object.

Example:

```
DATA TIME TYPE T VALUE '172545'.

WRITE TIME.
WRITE / TIME+2(2).
CLEAR TIME+2(4).
WRITE / TIME.
```

The output appears as follows:

```
172545

25

170000
```

First, the minutes are selected by specifying an offset in the WRITE statement. Then, the minutes and seconds are set to their initial values by specifying an offset in the clear statement.

•**Introduction**

•**Conversion Rules for Elementary Data Types**

•**Conversion Rules for References**

•**Conversion Rules for Structures**

•**Conversion Rules for Internal Tables**

•**Alignment of Data Objects**

Every time you assign a data object to a variable, the data types involved must either be compatible, that is, their technical attributes (data type, field length, number of decimal places) must be identical, or the data type of the source field must be convertible into the data type of the target field.

In ABAP, two non-compatible data types can be converted to each other if a corresponding conversion rule exists. If data types are compatible, no conversion rule is necessary.

For example, If you use the MOVE statement to transfer values between non-compatible objects, the value of the source object is always converted into the data type of the target object.

There are 64 possible type combinations between the elementary data types. ABAP supports automatic type conversion and length adjustment for all of them except type D (date) and type T (time) fields which cannot be converted into each other.

NOTE:

If you try to assign values between two data types for which no conversion rule exists, a syntax error or runtime error occurs.

ABAP currently uses class and interface variables within ABAP Objects. Both are pointers to objects. You can assign values to them in the following combinations:

- **If the two class references are incompatible, the class of the target field must be the predefined empty class OBJECT.**
- **When you assign a class reference to an interface reference, the class of the source field must implement the interface of the target field.**
- **If two interface references are incompatible, the interface of the target field must contain the interface of the source field as a component.**
- **When you assign an interface reference to a class reference, the class of the source field must be the predefined empty class OBJECT.**

ABAP has one rule for converting structures that do not contain internal tables as components. There are no conversion rules for structures that contain internal tables. You can only make assignments between structures that are compatible.

You can combine convertible structures in the following combinations:

- Converting a structure into a non-compatible structure
- Converting elementary fields into structures
- Converting structures into elementary fields

If you convert a structure into a shorter structure, the original structure is truncated. If you convert a structure into a longer one, the parts at the end are not initialized according to their type, but filled with blanks.

Example:

```
DATA: BEGIN OF FS1,
INT TYPE I VALUE 5,
PACK TYPE P DECIMALS 2 VALUE ‘2.26’,
TEXT(10) TYPE C VALUE ‘Fine text’,
FLOAT TYPE F VALUE ‘1.234e+05’,
DATA TYPE D VALUE ‘19950916’,
END OF FS1.
DATA: BEGIN OF FS2,
INT TYPE I VALUE 3,
PACK TYPE P DECIMALS 2 VALUE ‘72.34’,
TEXT(5) TYPE C VALUE ‘Hello’,
END OF FS2.
WRITE: / FS1-INT, FS1-PACK; FS1-TEXT, FS1-FLOAT, FS1-DATE.
WRITE: / FS2-INT, FS2-PACK, FS2-TEXT.
MOVE FS1 TO FS2.
WRITE: / FS2-INT, FS2-PACK, FS2-TEXT.
```

The output appears as follows:

```
 5 2.26 Fine text 1.234000000000000E+05 09161995
 3 72.34 Hello
 5 2.26 Fine
```

This example defines two different structures, FS1 and FS2. In each one, the first two components have the same data type. After assigning FS1 to FS2, only the result for the first two components is as if they had been moved component-by-component.
FS2-TEXT is filled with the first five characters of FS1-TEXT.
All other positions of FS1 are omitted.

Internal tables can only be converted into other internal tables. You cannot convert them into structures or elementary fields.

Internal tables are convertible if their line types are convertible. The convertibility of internal tables does not depend on the number of lines.

Conversion rules for internal tables:

- **Internal tables which have internal tables as their line type are convertible if the internal tables which define the line types are convertible.**
- **Internal tables which have line types that are structures with internal tables as components are convertible according to the conversion rules for structures if the structures are compatible.**

Elementary fields with types I and F occupy special memory addresses that are platform-specific. For example, the address of a type I field must be divisible by 4, and the address of a type F field by 8. Consequently, type I and F fields are known as aligned fields. Structures containing fields with type I or F are also aligned, and may contain filler fields immediately before their aligned components.

The system normally aligns fields and structures automatically when you declare them.

You must take alignment into account in the following cases:

- •When you pass elementary fields or structures to a procedure as actual parameters where the corresponding formal parameter is not typed accordingly.
- •When you declare field symbols.
- •When you use a work area with an ABAP Open SQL statement that does not have the same type as the database table as defined in the ABAP Dictionary.
- •When you process components of structures.

Now you have learnt how to…

- Assign Values using different forms of MOVE and WRITE TO statements
- Reset data objects to Initial Values using CLEAR statement
- Perform Numerical Operations
- Process Character Strings
- Specify Offset Values for Data Objects
- Possible Type Conversions

TASK : Define a Field String Address with FirstName value Fred ,Surname value Flintstone, Initials value ‘FF’,Street ‘Cave Avenue’,Number value ‘11’,PostCode value’98765’, City value BedRock as components. Define another Field String Name with FirstName,Initials,Title as components. Send the value of corresponding fields to Name.

Output: In the example , the values of NAME-SURAME, NAME-FIRSTNAME and NAME-INITIALS are set to ‘Flintstone’, ‘Fred’, and ‘FF’. NAME-TITLE retains the value ‘Mister’.

```
DATA:     BEGIN OF ADDRESS,
                    FIRSTNAME(20) VALUE ‘FRED’,
                    SURNAME(20)    VALUE ‘Flintstone’,
                    INITIALS(4)        VALUE ‘FF’,
                    STREET(20)       VALUE ‘Cave Avenue’,
                    NUMBER TYPE I VALUE ‘11’,
                    POSTCODE TYPE N VALUE ‘98765’,
                    CITY(20)       VALUE ‘Bedrock’.
     END OF ADDRESS.
DATA:     BEGIN OF NAME,
                    FIRSTNAME(20),        SURNAME(20),
                    INITIALS(4),
                    TITLE(10)  VALUE ‘Mister’,
          END OF NAME.
     MOVE-CORRESPONDING ADDRESS TO NAME.
```

In the example , the values of NAME-SURAME, NAME-FIRSTNAME and NAME-INITIALS are set to ‘Flintstone’, ‘Fred’, and ‘FF’. NAME-TITLE retains the value ‘Mister’.

Task2: String Exercise as per output.

Output: The first WRITE statement writes the first 6 positions of the field NUMBER left-justified to the last 12 positions of the field STRING. The second WRITE statement writes the first 6 positions of the field NUMBER centered to the last 12 positions of the field STRING. The third WRITE statement writes the first 6 positions of the field NUMBER right-justified to the last 12 positions of the field STRING.
This produces the following output:

```
123456

   123456

      123456
```

Solution:

```
Data:        STRING(20),

     NUMBER(8)    TYPE   C        VALUE          '123456',
     OFFSET       TYPE   I        VALUE 8,
     LENGTH       TYPE   I        VALUE 12.

WRITE NUMBER+(6) TO STRING +OFFSET(LENGTH) LEFT-
JUSTIFIED.

WRITE: / STRING.
CLEAR    STRING.

WRITE NUMBER+(6) TO STRING +OFFSET(LENGTH)
CENTERED.

WRITE: / STRING.  CLEAR    STRING.

WRITE NUMBER+(6) TO STRING +OFFSET(LENGTH) RIGHT-
JUSTIFIED.

WRITE: / STRING.
CLEAR    STRING.
```

Task: Shifting a Structure up to given string.Show following output.

Output: This produces the following output:

```
abcdefghij
efghij
efghij
   abcdef
efghijabcd
```

Solution

```
DATA: T(10) VALUE 'abcdefghij',
STRING LIKE T,
STR(2) VALUE 'ef'.
STRING = T.
WRITE STRING.
        SHIFT STRING UP TO STR.
WRITE / STRING.
STRING = T.
SHIFT STRING UP TO STR LEFT.
WRITE / STRING.
STRING = T.
SHIFT STRING UP TO STR RIGHT.

WRITE / STRING.
STRING = T.
SHIFT STRING UP TO STR CIRCULAR.
WRITE / STRING.
```

Task4: Replacing Field Contents.

Output:

The output appears as follows:

```
abcdefghij

abklmnghij

abklmnefgh

abklghij

abklmnopgh
```

Note how, in the last line, the field STRING is truncated on the right. The search pattern 'cdef' of length 4 is replaced by 'klmnop' of length 6. Then, the rest of the field STRING is filled up to the end of the field.

Solution 4:

```
DATA: T(10) VALUE 'abcdefghij',
STRING LIKE T,
STR1(4) VALUE 'cdef',
STR2(4) VALUE 'klmn',
STR3(2) VALUE 'kl',
STR4(6) VALUE 'klmnop',
LEN TYPE I VALUE 2.

    STRING = T.
    WRITE STRING.

    REPLACE STR1 WITH STR2 INTO STRING.
    WRITE / STRING.

    STRING = T.
    REPLACE STR1 WITH STR2 INTO STRING LENGTH LEN.
    WRITE / STRING.

    STRING = T.
    REPLACE STR1 WITH STR3 INTO STRING.
    WRITE / STRING.

    STRING = T.
    REPLACE STR1 WITH STR4 INTO STRING.
    WRITE / STRING.
```

TASK5: Finding Character strings

Output: The output appears as follows:

```
This is a fast first example.

SY-FDPOS:    10

This is a fast FIRST example.

SY-FDPOS:    4

Off:        15
```

Note that in order to find the second word containing 'ft' after finding the word 'fast', you have to add 2 to the offset SY-FDPOS and start the search at the position POS. Otherwise, the word 'fast' would be found again. To obtain the offset of 'first' in relation to the start of the field STRING, it is calculated from POS and SY-FDPOS.

Solution 5

```
DATA: STRING(30) VALUE 'This is a fast first
example.',
POS TYPE I,
OFF TYPE I.
WRITE / STRING.

SEARCH STRING FOR 'ft' ABBREVIATED.
WRITE: / 'SY-FDPOS:', SY-FDPOS.
      POS = SY-FDPOS + 2.
SEARCH STRING FOR 'ft' ABBREVIATED
STARTING AT POS AND MARK.
WRITE / STRING.
WRITE: / 'SY-FDPOS:', SY-FDPOS.
OFF = POS + SY-FDPOS -1.
WRITE: / 'Off:', OFF.
```

Task6: Condensing Field Contents

Output:

The output appears as follows:

```
one two three four !
                  Length:          25
                                 one two three four !
                                                   Length:          18

onetwothreefour !
                   Length:          15
```

Note that the total length of the field STRING remains unchanged, but that the deleted blanks appear again on the right.

Solution6:

```
DATA: STRING(25) VALUE ' one two three four',
LEN TYPE I.
LEN = STRLEN( STRING ).
WRITE: STRING, '!'.
WRITE: / 'Length: ', LEN.
        CONDENSE STRING.
LEN = STRLEN( STRING ).
WRITE: STRING, '!'.
WRITE: / 'Length: ', LEN.
        CONDENSE STRING NO-GAPS.
LEN = STRLEN( STRING ).
WRITE: STRING, '!'.
WRITE: / 'Length: ', LEN.
```

Task7: Conversion Rule for Structures

Output: appears as follows:

```
 5 2.26 Fine text 1.2340000000000000E+05 09161995
  3 72.34 Hello
  5 2.26 Fine
```

This example defines two different structures, FS1 and FS2. In each one, the first two components have the same data type. After assigning FS1 to FS2, only the result for the first two components is as if they had been moved component-by-component.
FS2-TEXT is filled with the first five characters of FS1-TEXT.
All other positions of FS1 are omitted.

Solution7:

```
DATA: BEGIN OF FS1,
INT TYPE I VALUE 5,
PACK TYPE P DECIMALS 2 VALUE ‘2.26’,
TEXT(10) TYPE C VALUE ‘Fine text’,
FLOAT TYPE F VALUE ‘1.234e+05’,
DATA TYPE D VALUE ‘19950916’,
END OF FS1.
DATA: BEGIN OF FS2,
INT TYPE I VALUE 3,
PACK TYPE P DECIMALS 2 VALUE ‘72.34’,
TEXT(5) TYPE C VALUE ‘Hello’,
END OF FS2.
WRITE: / FS1-INT, FS1-PACK; FS1-TEXT, FS1-FLOAT, FS1-DATE.
WRITE: / FS2-INT, FS2-PACK, FS2-TEXT.
MOVE FS1 TO FS2.
WRITE: / FS2-INT, FS2-PACK, FS2-TEXT.
```