ABAP Dynamic Programming

# Field Symbols

Field symbol is a placeholder for data object, which points to the value present at the memory address of a data object. It does not reserve any physical memory space when we declare them. It only points to a data object at run time. Field symbols are of two types:

* Typed Field Symbol
* Generic Field Symbol

**Typed Field Symbol** – Typed field symbol can be declared as:

DATA: var TYPE i VALUE 2.

FIELD-SYMBOLS: <fs\_num> TYPE i.

ASSIGN var TO <fs\_num>.

WRITE: / <fs\_num>.

<fs\_num> = 4.

WRITE: / var.

The output will be 2 and 4.

**NOTE:**

* Typed field symbols can point to the data objects of specified type only.
* After assigning a data object to a field symbol, if we make any changes to the field symbol value, then the value of corresponding data object is also updated.

**Field symbol as a replacement of Work area:**

**Modifying internal table records** – We can declare a field symbol of type any structure, which we can use while looping through an internal table.

DATA: lt\_mara TYPE STANDARD TABLE OF mara.

FIELD-SYMBOLS: <fs\_mara> TYPE mara.

SELECT \* FROM mara INTO TABLE lt\_mara UP TO 10 ROWS.

LOOP AT lt\_mara ASSIGNING <fs\_mara>.

<fs\_mara>-matkl = 'DEMO'.

ENDLOOP.

**NOTE:**

* If we change any field of structure in field symbol, the corresponding field in internal will get updated. We don’t need to write the **MODIFY** statement which we would have written if we had used work area. This is because work area stores a copy of the internal table row, whereas field symbol directly references the internal table row.
* Hence processing of internal table with field symbol is faster than the processing of internal table with work area.

**Generic Field Symbol:**

Dynamic programming is actually implemented using generic field symbols. The most commonly used generic types are **TYPE ANY** and **TYPE ANY TABLE**.

FIELD-SYMBOLS: <fs\_str> TYPE ANY.

FIELD-SYMBOLS: <fs\_tab> TYPE ANY TABLE.

Here we can assign any data object to **TYPE ANY** field symbol whereas **TYPE ANY TABLE** field symbol is used for assigning any internal table.

**TYPE ANY:**

Let us assign a work area of type **MARA** to a TYPE ANY field symbol and then populate the work area using field symbol.

FIELD-SYMBOLS: <fs\_str> TYPE ANY.

FIELD-SYMBOLS: <fs\_data> TYPE ANY.

DATA: lw\_mara TYPE mara.

ASSIGN lw\_mara TO <fs\_str>.

IF <fs\_str> IS ASSIGNED.

ASSIGN COMPONENT 'MATNR' OF STRUCTURE <fs\_str> TO <fs\_data>.

IF <fs\_data> IS ASSIGNED.

<fs\_data> = 'MAT001'.

UNASSIGN <fs\_data>.

ENDIF.

UNASSIGN <fs\_str>.

ENDIF.

**NOTE:**

* After assigning **lw\_mara** to **<fs\_str>**, we cannot directly use the ‘-‘ operator on field symbol to access the fields of MARA structure i.e. **<fs\_str>-matnr** would produce syntax error. This is because the field symbol type is declared only at runtime not at compile time.
* So to access the **matnr** field with field symbol, first we need to assign that field component to a different field symbol and then use the new field symbol to update the **matnr**field as show in above code snippet.
* After execution of above code snippet, the value of **lw\_mara-matnr** would be **MAT001**.

**TYPE ANY TABLE:**

We can assign any internal table to this field symbol. Let us analyze the below code snippet to understand how we could use such field symbol.

FIELD-SYMBOLS: <fs\_tab> TYPE ANY TABLE.

FIELD-SYMBOLS: <fs\_str> TYPE any.

FIELD-SYMBOLS: <fs\_data> TYPE any.

DATA: lt\_mara TYPE STANDARD TABLE OF mara.

DATA: lw\_mara TYPE mara.

ASSIGN lt\_mara TO <fs\_tab>.

SELECT \* FROM mara INTO TABLE lt\_mara UP TO 10 ROWS.

LOOP AT <fs\_tab> ASSIGNING <fs\_str>.

IF <fs\_str> IS ASSIGNED.

ASSIGN COMPONENT 'MATKL' OF STRUCTURE <fs\_str> TO <fs\_data>.

IF <fs\_data> IS ASSIGNED.

IF <fs\_data> EQ '01'.

\*\*\*\*\*\*\*\*\*\*\* Do some processing \*\*\*\*\*\*\*\*\*

ENDIF.

UNASSIGN <fs\_data>.

ENDIF.

ENDIF.

ENDLOOP.

**Reading internal table using generic field symbol:**

FIELD-SYMBOLS: <fs\_tab> TYPE ANY TABLE.

FIELD-SYMBOLS: <fs\_str> TYPE any.

DATA: lt\_mara TYPE STANDARD TABLE OF mara.

ASSIGN lt\_mara TO <fs\_tab>.

SELECT \* FROM mara INTO TABLE lt\_mara UP TO 10 ROWS.

READ TABLE <fs\_tab> ASSIGNING <fs\_str> WITH KEY ('MATNR') = 'MAT001'.

**NOTE:**

* Since **<fs\_tab>** is a generic field symbol, its type will be known only at runtime, hence we cannot directly write the fields of MARA structure after WITH KEY, instead we have to write the field name within parenthesis as shown above.
* In ABAP, this parenthesis indicates the compiler that the value of the operand will be decided at runtime, hence we don’t get any compilation error.

# Data Reference

According to SAP documentation, **Data references can point to any data objects or to their parts (components, rows of internal tables, or sections specified by offsets and lengths).**

So data references are nothing but pointers. It stores the memory address of any data object. But to access the actual data object which data reference is pointing to, we first need to deference it using dereferencing operator **->\***.

**Difference between field symbol and data reference:**

Field symbol is a placeholder for data object to which it is assigned and points to the content of data object hence it can be used at any operand position (no need to dereference it) and works with the content of the referenced memory area (value semantics).

Data references are pointers to data objects and it contains the memory address of data object (reference semantics). Data reference cannot be used at operand position directly; it should be dereferenced first.

**Working with data reference:**

There can be two types of data references:

* Typed Data Reference
* Generic Data Reference

**Typed Data Reference:**

Typed data reference variable can be declared as:

DATA lr\_num TYPE REF TO i.

CREATE DATA lr\_num.

Here first statement declares a reference variable **lr\_num** which can point to any data object of type **“i”**. And second statement creates an anonymous data object of type **“i** ”and assigns the reference of this data object to **lr\_num**. Now if we want to change the value of data object, then it can be done by dereferencing **lr\_num** by using dereference operator **->\*** as shown below:

DATA lr\_num TYPE REF TO i.

CREATE DATA lr\_num.

lr\_num->\* = 2.

WRITE: / lr\_num->\*.

The output will be 2.

**Assigning existing data object to data reference:**

If you want to assign the reference of an existing data object to a data reference, you can use **GET REFERENCE** statement.

DATA: lv\_num TYPE i VALUE 2.

DATA: lr\_num TYPE REF TO i.

GET REFERENCE OF lv\_num INTO lr\_num.

lr\_num->\* = 4.

WRITE: / lv\_num.

Here **lv\_num** is an existing data object (not anonymous data object). The output would be 4.

**Working with structures:**

DATA: lr\_mara TYPE REF TO mara.

CREATE DATA lr\_mara.

lr\_mara->matnr = '1111'.

lr\_mara->matkl = '03'.

Here individual components of the structure can be accessed with **->** operator on data reference variable.

**Working with internal tables:**

While processing internal table row, we can use **REFERENCE INTO** statement to set references to table rows as shown below:

DATA: lr\_mara TYPE REF TO mara.

DATA: lt\_mara TYPE TABLE OF mara.

SELECT \* FROM mara INTO TABLE lt\_mara UP TO 10 ROWS.

LOOP AT lt\_mara REFERENCE INTO lr\_mara.

WRITE: / lr\_mara->matnr.

ENDLOOP.

**Generic Data Reference:**

Generic data reference can be declared as:

DATA: lr\_num TYPE REF TO data.

CREATE DATA lr\_num TYPE i.

Here first statement declares a generic data reference **lr\_num** which can point to any data object. And second statement creates an anonymous data object of type “i” and assigns its reference to **lr\_num**.

‘**data**’ in ABAP is a generic data type.

Now since **lr\_num** is generic**, lr\_num->\*** cannot be directly used at operand position. Hence the below statement would not be allowed.

lr\_num->\* = 2.

**So in case of generic data reference, it can only be dereferenced using a field symbol**, and this field symbol can be used at any operand position to manipulate the value of data object as shown below:

DATA: lr\_num TYPE REF TO data.

FIELD-SYMBOLS: <num> TYPE any.

CREATE DATA lr\_num TYPE i.

ASSIGN lr\_num->\* TO <num>.

<num> = 3.

**NOTE:**

* After **ASSIGN** statement you should check **sy-subrc** If field symbol assignment is successful, **sy-subrc** will be 0 otherwise it will be 4.

**Working with structures:**

DATA: lr\_str TYPE REF TO data.

FIELD-SYMBOLS: <str> TYPE any.

FIELD-SYMBOLS: <data> TYPE any.

CREATE DATA lr\_str TYPE mara.

ASSIGN lr\_str->\* TO <str>.

ASSIGN COMPONENT 'MATNR' OF STRUCTURE <str> TO <data>.

<data> = '112'.

Here **CREATE DATA** statement creates an anonymous data object (MARA structure) and assigns its reference to the generic data reference **lr\_str**, which then can be dereferenced into a generic field symbol **<str>**. Now, to access individual component of MARA structure, **ASSIGN COMPONENT** statement can be used.

**Dynamically create data objects:**

**Requirement:** Selection screen parameter “**Table Name**” will take a table name as input and display the corresponding table entries as output.

**Solution:**

PARAMETERS: p\_tname TYPE tabname.

DATA: lr\_tab TYPE REF TO data.

FIELD-SYMBOLS: <tab> TYPE ANY TABLE.

CREATE DATA lr\_tab TYPE TABLE OF (p\_tname).

ASSIGN lr\_tab->\* TO <tab>.

IF sy-subrc EQ 0.

SELECT \* FROM (p\_tname) INTO TABLE <tab> UP TO 10 ROWS.

cl\_demo\_output=>display( <tab> ).

ENDIF.

**Explanation:**

Here **lr\_tab** is a generic data reference and **<tab>** is a generic field symbol for internal table. In **CREATE DATA** statement, the type of data object is mentioned in parenthesis which means that the type will be determined at runtime based on the value of parameter **p\_tname**. After that we have dereferenced the data reference **lr\_tab** into a generic field symbol **<tab>.** Now this field symbol can be used to do any valid operation on the internal table.

**Difference between data reference and object reference:**

There are two types of reference variable:

* Data reference and
* Object reference

Data reference variable can store the reference to any data object (variable, structures, internal tables etc.) whereas Object reference variable can store the reference to any class object.

For data reference variables, either the generic data type or a completely specified data type can be specified. For object reference variables, either a class or an interface can be specified.

# ABAP RTTS

Now here we will see one example of dynamic programming approach and also a brief introduction to ABAP RTTS.

ABAP **Runtime Type Services (RTTS)** consists of two components:

* **Runtime Type Identification (RTTI)** – Provides the methods to get the type definition of data objects at runtime.
* **Runtime Type Creation (RTTC)** – Provides the methods to create the data objects at runtime with any type definition.

Basically, ABAP RTTS provides a set of classes, whose methods can be used for runtime type identification and runtime type creation

**An example of dynamic programming:**

**Requirement:** As an ABAP developer, very often we get the situation where we need to write data from an internal table to a file on application server.

**Solution:** We will build one class having a method which will take any internal table as input and write its content in a file on application server.

Class Definition:

CLASS cl\_appserver\_writer DEFINITION.

PUBLIC SECTION.

CLASS-METHODS: write IMPORTING

iv\_filename TYPE string

it\_data TYPE ANY TABLE

write\_header TYPE abap\_bool DEFAULT space

EXPORTING

ev\_message TYPE string.

ENDCLASS.

Here importing parameter **it\_data** is of **TYPE ANY TABLE** so that it can receive any internal table.

Class Implementation:

CLASS cl\_appserver\_writer IMPLEMENTATION.

METHOD write.

TYPES: BEGIN OF ty\_comp\_detail,

name TYPE abap\_compname,

descr TYPE scrtext\_m,

END OF ty\_comp\_detail.

DATA: lo\_type\_def TYPE REF TO cl\_abap\_typedescr.

DATA: lo\_struct\_def TYPE REF TO cl\_abap\_structdescr.

DATA: lo\_table\_def TYPE REF TO cl\_abap\_tabledescr.

DATA: lo\_data\_def TYPE REF TO cl\_abap\_datadescr.

DATA: lo\_element\_def TYPE REF TO cl\_abap\_elemdescr.

DATA: lt\_components TYPE abap\_compdescr\_tab.

DATA: wa\_components LIKE LINE OF lt\_components.

DATA: lv\_str TYPE string.

DATA: lv\_filerow TYPE string.

DATA: lv\_counter TYPE i VALUE 0.

DATA: lw\_field\_info TYPE dfies.

DATA: ls\_comp\_detail TYPE ty\_comp\_detail.

DATA: lt\_comp\_detail TYPE TABLE OF ty\_comp\_detail.

FIELD-SYMBOLS: <row> TYPE any.

FIELD-SYMBOLS: <field\_value> TYPE any.

\* Using RTTS to get the runtime type information of the internal table

lo\_type\_def = cl\_abap\_tabledescr=>describe\_by\_data( it\_data ).

lo\_table\_def ?= lo\_type\_def.

lo\_data\_def = lo\_table\_def->get\_table\_line\_type( ).

lo\_struct\_def ?= lo\_data\_def.

\* Get the components of the structure

lt\_components = lo\_struct\_def->components.

CLEAR: lo\_data\_def.

\* If the WRITE\_HEADER is ABAP\_TRUE then fetch the label

\* of data element associated to each component of the

\* line type structure of internal table, if no data element

\* is associated then use component name as the header text

IF write\_header EQ abap\_true.

LOOP AT lt\_components INTO wa\_components.

lo\_data\_def = lo\_struct\_def->get\_component\_type( wa\_components-name ).

lo\_element\_def ?= lo\_data\_def.

lw\_field\_info = lo\_element\_def->get\_ddic\_field( ).

ls\_comp\_detail-name = lw\_field\_info-rollname. "Get the data element name

\* Calling FM to get data element text

CALL FUNCTION 'WCGW\_DATA\_ELEMENT\_TEXT\_GET'

EXPORTING

i\_data\_element = lw\_field\_info-rollname

i\_language = sy-langu

IMPORTING

e\_scrtext\_m = ls\_comp\_detail-descr

EXCEPTIONS

error = 1.

IF ls\_comp\_detail-descr IS INITIAL.

ls\_comp\_detail-descr = wa\_components-name.

ENDIF.

APPEND ls\_comp\_detail TO lt\_comp\_detail.

CLEAR: ls\_comp\_detail.

ENDLOOP.

ENDIF.

OPEN DATASET iv\_filename FOR OUTPUT IN TEXT MODE ENCODING DEFAULT.

IF sy-subrc EQ 0.

\* Writing header text for each column separated by comma

IF write\_header EQ abap\_true.

LOOP AT lt\_comp\_detail INTO ls\_comp\_detail.

lv\_counter = lv\_counter + 1.

IF lv\_counter EQ 1.

lv\_filerow = ls\_comp\_detail-descr.

ELSE.

CONCATENATE lv\_filerow ',' ls\_comp\_detail-descr INTO lv\_filerow.

ENDIF.

ENDLOOP.

TRANSFER lv\_filerow TO iv\_filename.

CLEAR: lv\_filerow, lv\_counter.

ENDIF.

\* Writing internal table content separated by comma

LOOP AT it\_data ASSIGNING <row>.

LOOP AT lt\_components INTO wa\_components.

lv\_counter = lv\_counter + 1.

ASSIGN COMPONENT wa\_components-name OF STRUCTURE <row> TO <field\_value>.

IF <field\_value> IS ASSIGNED.

lv\_str = <field\_value>.

IF lv\_counter EQ 1.

lv\_filerow = lv\_str.

ELSE.

CONCATENATE lv\_filerow ',' lv\_str INTO lv\_filerow.

ENDIF.

UNASSIGN <field\_value>.

ENDIF.

ENDLOOP.

TRANSFER lv\_filerow TO iv\_filename.

CLEAR: lv\_filerow, lv\_counter.

ENDLOOP.

CLOSE DATASET iv\_filename.

ev\_message = 'Success'.

ELSE.

ev\_message = 'Failure'.

ENDIF.

ENDMETHOD.

ENDCLASS.

Here the classes **CL\_ABAP\_\*DESCR** are provided by the ABAP RTTS and used to get the type definition of data objects at runtime. Also we have extracted the data element name of each component of line type structure of internal table **it\_data** using RTTS classes. Then we fetched the data element label using the FM **WCGW\_DATA\_ELEMENT\_TEXT\_GET**. This label is used to write the header for each column of internal table **it\_data** if WRITE\_HEADER parameter of class is provided with ABAP\_TRUE.

**Using the Class** – The above designed class can be used as:

DATA: lt\_data TYPE STANDARD TABLE OF mara.

DATA: lv\_filename TYPE string.

DATA: lv\_message TYPE string.

SELECT \* FROM mara INTO TABLE lt\_data UP TO 5 ROWS.

cl\_appserver\_writer=>write(

EXPORTING

iv\_filename = 'D:\usr\sap\testdata.csv'

it\_data = lt\_data

write\_header = abap\_true

IMPORTING

ev\_message = lv\_message

).

WRITE: / lv\_message.

Here we are passing one internal table of structure MARA to the class, and subsequently its content will be written on application server as comma separated values. However, we can pass internal table of any structure. This file can also be downloaded from application server to an excel spreadsheet.

So this is how field symbol, data reference, generic data type, RTTS helps in dynamic programming approach.