Structured Text (ST)

CoDeSys

Smart Software Solutions

We software Automation.

Structured Text (ST)

is one of the 5 languages supported by the IEC 61131-3 standard.

Smart Software Solutions

We software Automation.

Introduction

After this module you will be …

- familiar with the IEC 61131-3 language “Structured Text”,
- the language syntax and
- able to read and write POUs in the new powerful language

Some words about ST …

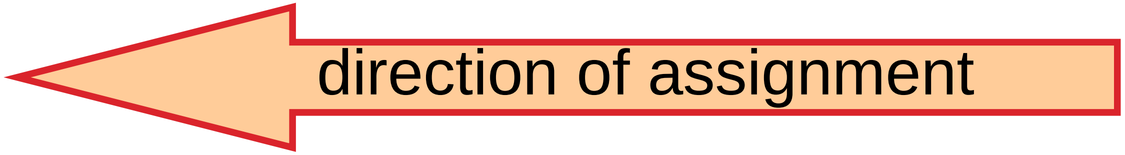
- textual language
- high-level language
- PASCAL- like
- Most suitable IEC 61131-3 languages for …
  - conditional programming
  - programming loops
- A few years ago not so well known in the PLC world,
  => now it is becoming more and more popular

Language Elements

Assignment

don‘t forget the “;”
at the end

```
Result := 5 + 3 ;
```



direction of assignment

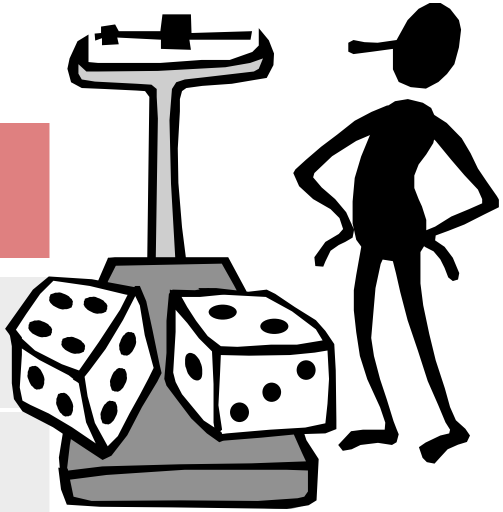
Analog

IL, FBD, LD	ST
ADD	+
SUB	-
MUL	*
DIV	/
MOD	**MOD**

Language Elements

Analog

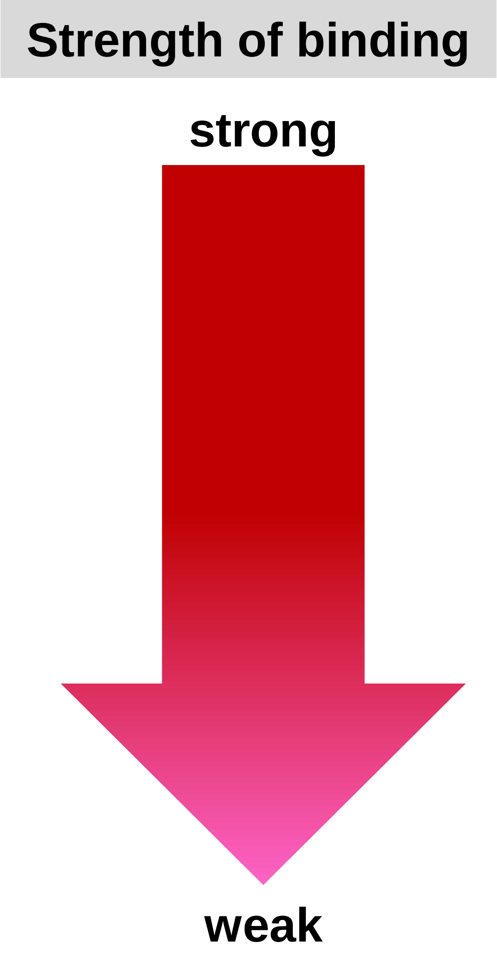
IL, FBD, LD	ST
EQ	=
NE	<>
GE	>=
GT	>
LE	<=
LT	<



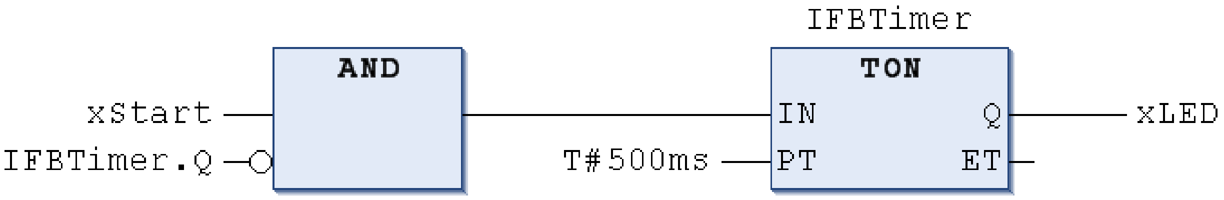
Language Elements

Operators

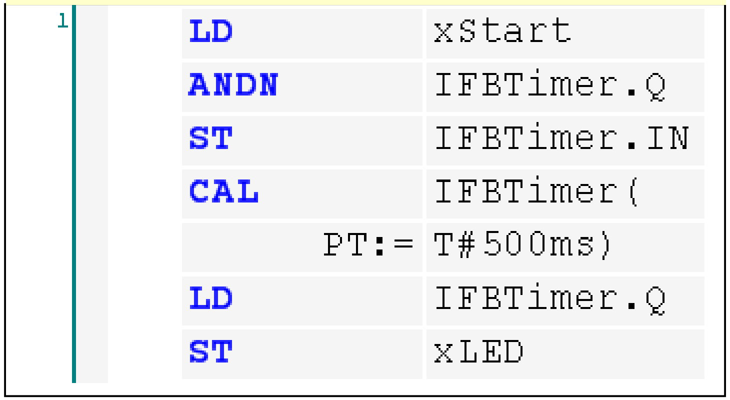
Operation	Symbol
to bracket	( )
Function call	FunctionName(Parameter)
Exponentiation	EXPT
Binary complement	NOT
Multiplication	*
Division	/
Modulo	MOD
Addition	+
Substraction	-
Comparison	<, >, <=, >=, =, <>
AND	AND
XOR	XOR
OR	OR



Using a function block

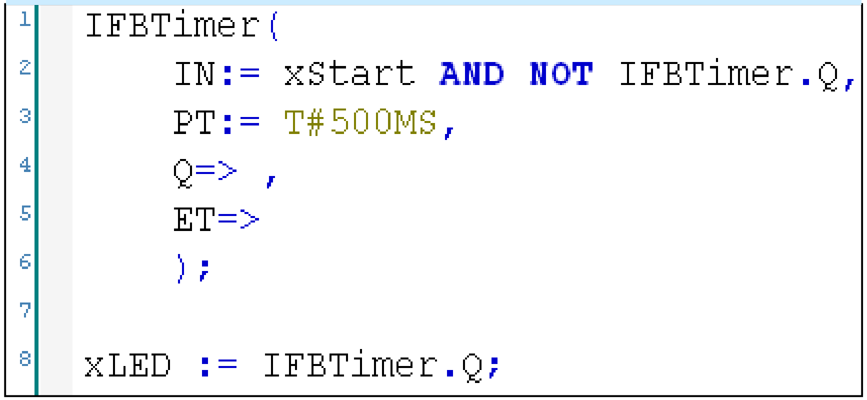


IL



```
LD      xStart
ANDN    IFBTimer.Q
ST      IFBTimer.IN
CAL     IFBTimer(
           PT:= T#500ms)
LD      IFBTimer.Q
ST      xLED
```

ST



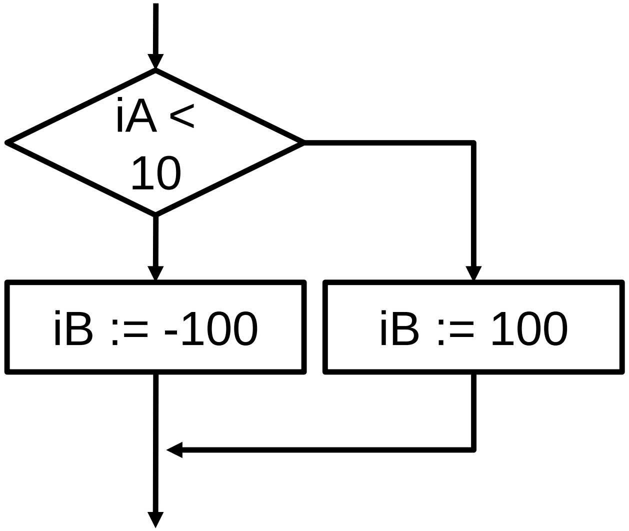
```
IFBTimer(
    IN:= xStart AND NOT IFBTimer.Q,
    PT:= T#500MS,
    Q=> ,
    ET=>
    );

xLED := IFBTimer.Q;
```

Conditional execution (IF)

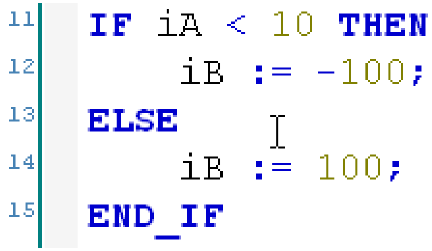
```
IF <Boolean_Expression> Then
      <instruction(s)>;
ELSIF <Boolean_Expression> Then
      <instruction(s)>;
ELSIF <Boolean_Expression> Then
      <instruction(s)>;
ELSE
      <instruction(s)>;
END_IF
```

optional

Conditional execution (IF)

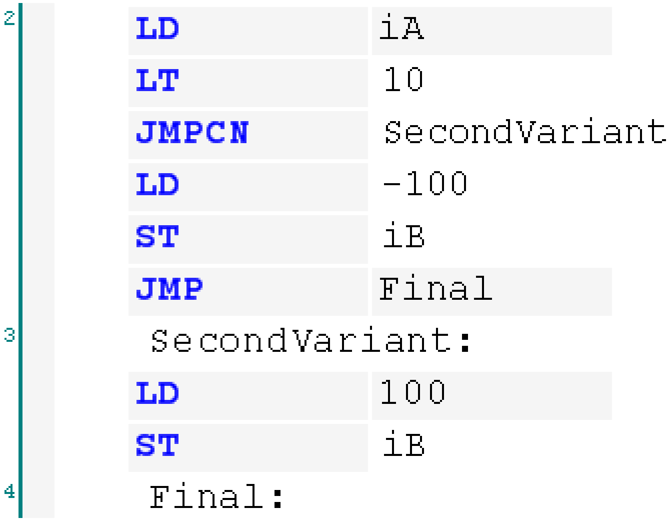


ST



```
IF iA < 10 THEN
    iB := -100;
ELSE
    iB := 100;
END_IF
```

IL



```
LD      iA
LT      10
JMPCN   SecondVariant
LD      -100
ST      iB
JMP     Final
 SecondVariant:
LD      100
ST      iB
 Final:
```

Conditional execution (CASE)

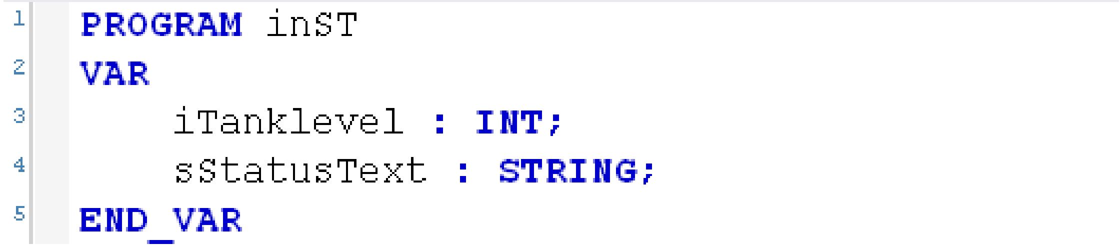
```
CASE <Var1> OF
<ValueA>:
        <Instruction(s)>;
<ValueB>:
        <Instruction(s)>;
<ValueC,ValueD,ValueE>:
        <Instruction(s)>;
<ValueF..ValueK>:
        <Instruction(s)>;
<ValueN>:
        <Instruction(s)>;
ELSE
        <Instruction(s)>;
END_CASE
```

integer data type

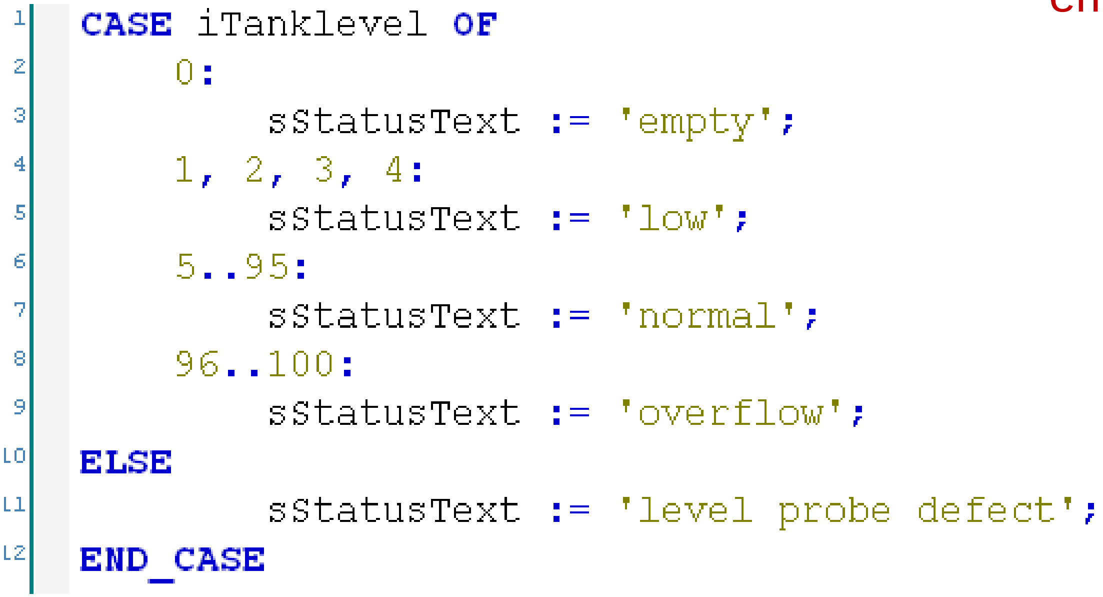
constant

optional

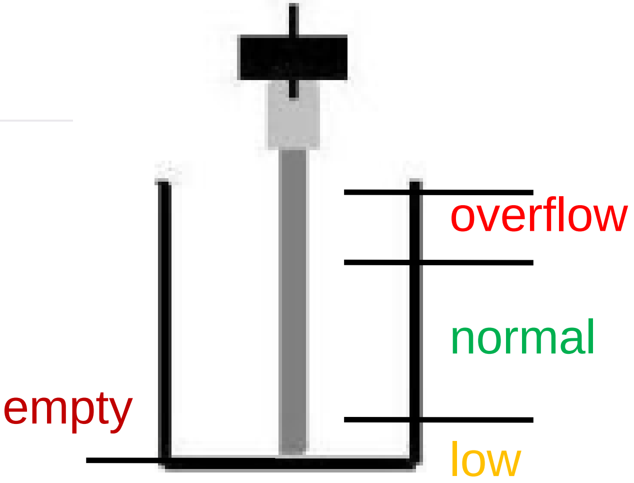
Conditional execution (CASE)



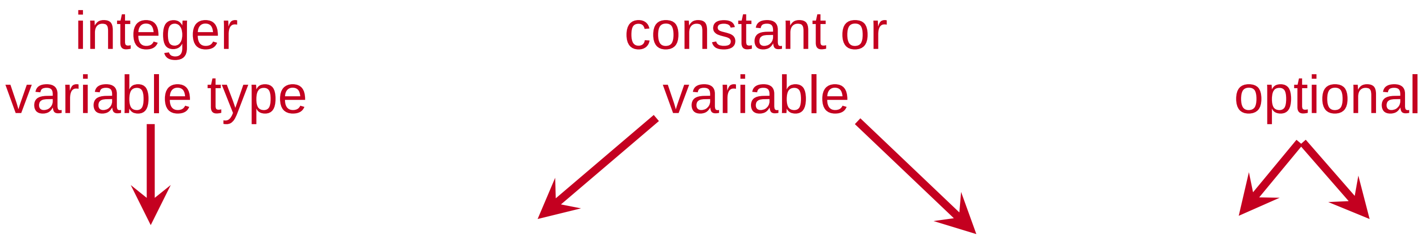
```
PROGRAM inST
VAR
    iTanklevel  : INT;
    sStatusText  : STRING;
END_VAR
```



```
CASE iTanklevel OF
    0:
        sStatusText := 'empty';
    1, 2, 3, 4:
        sStatusText := 'low';
    5..95:
        sStatusText := 'normal';
    96..100:
        sStatusText := 'overflow';
ELSE
        sStatusText := 'level probe defect';
END_CASE
```



Loop (FOR)



integer
variable type

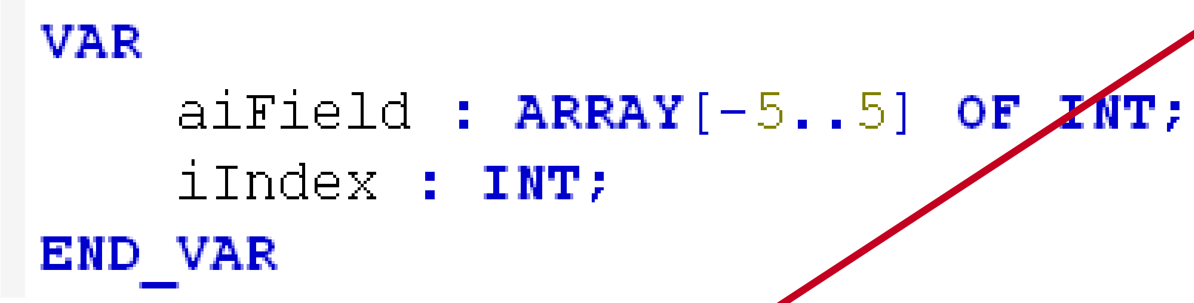
constant or
variable

optional

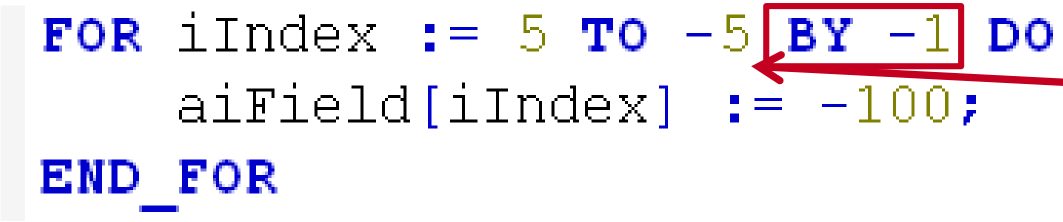
```
FOR <Var1>:= <INIT_Value> TO <END_Value> BY <Stepwidth>
DO
      <Instruction(s)>;
END_FOR
```

Loop (FOR)

Initializing an array

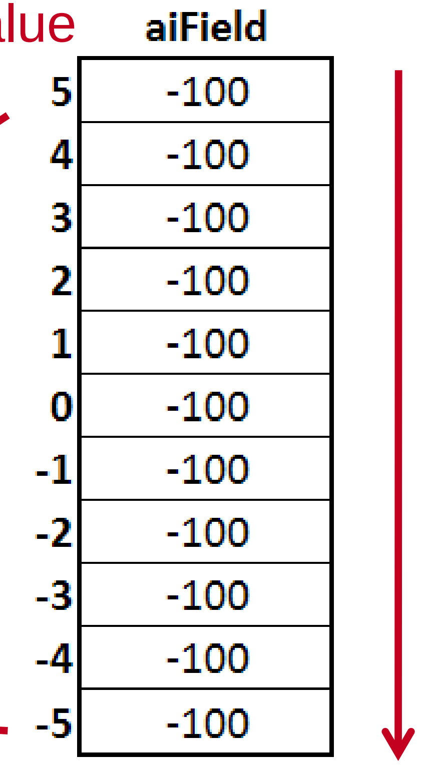


```
VAR
    aiField : ARRAY[-5..5] OF INT;
    iIndex : INT;
END_VAR
```



```
FOR iIndex := 5 TO -5 BY -1 DO
    aiField[iIndex] := -100;
END_FOR
```

step width is -1
instead of +1



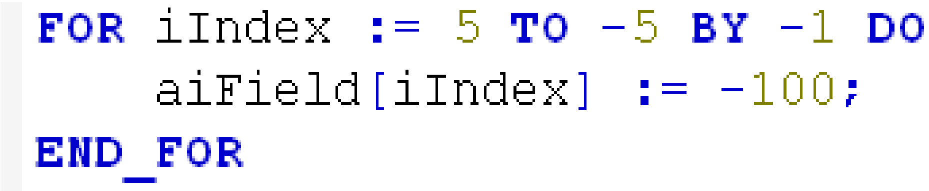
first value

	aiField
5	-100
4	-100
3	-100
2	-100
1	-100
0	-100
-1	-100
-2	-100
-3	-100
-4	-100
-5	-100

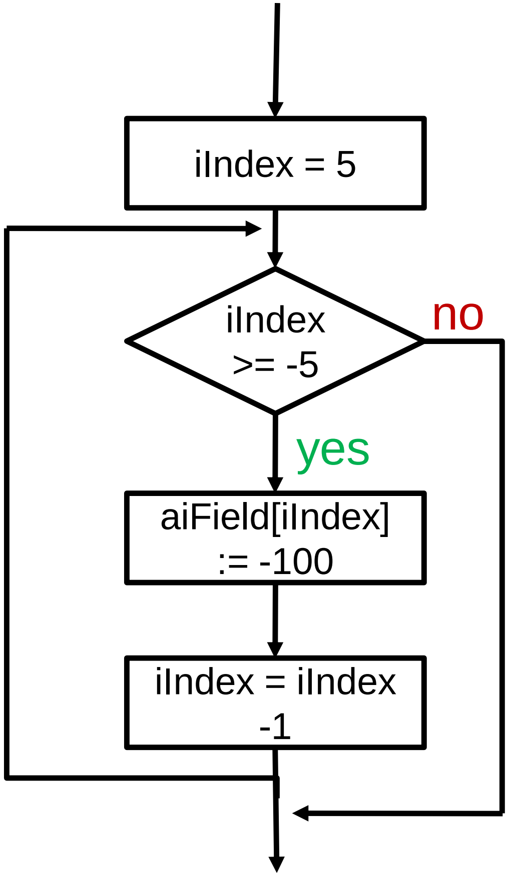
last value

Loop (FOR)

How does it work?



```
FOR iIndex := 5 TO -5 BY -1 DO
    aiField[iIndex] := -100;
END_FOR
```



Loop (WHILE)

```
WHILE <BooleanExpression> DO
    <Instruction(s)>;
END_WHILE
```

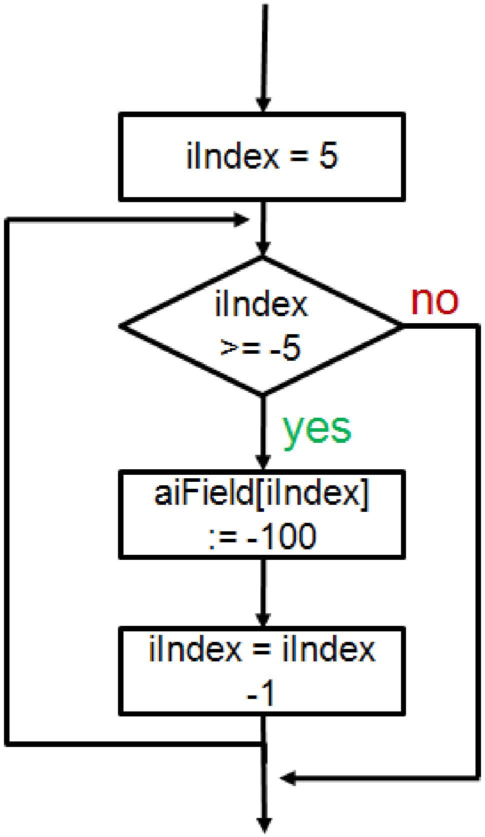
Loop (WHILE)

Initialization with a WHILE loop

	aiField
5	-100
4	-100
3	-100
2	-100
1	-100
0	-100
-1	-100
-2	-100
-3	-100
-4	-100
-5	-100

same behaviour

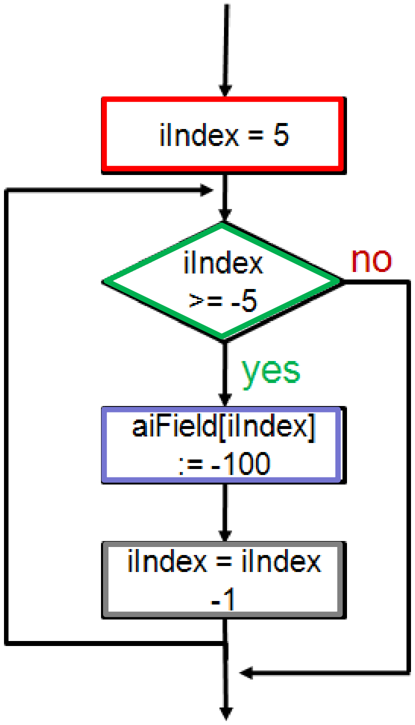
like a FOR loop



Loop (WHILE)

	aiField
5	-100
4	-100
3	-100
2	-100
1	-100
0	-100
-1	-100
-2	-100
-3	-100
-4	-100
-5	-100

How does it work?



```
iIndex := 5;
WHILE iIndex >= -5 DO
    aiField[iIndex] := -100;
    iIndex := iIndex - 1;
END_WHILE
```

Loop (REPEAT)

We go through the instructions once and then we check

```
REPEAT
       <Instruction(s)>;
UNTIL <BooleanExpression>
END_REPEAT
```

We stay in the loop UNTIL the expression is TRUE

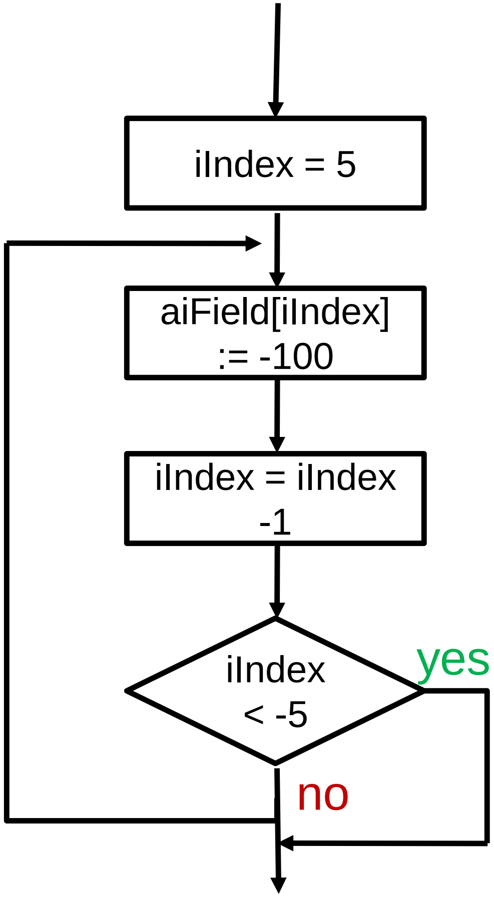
Loop (REPEAT)

Initialization with a REPEAT loop

	aiField
5	-100
4	-100
3	-100
2	-100
1	-100
0	-100
-1	-100
-2	-100
-3	-100
-4	-100
-5	-100

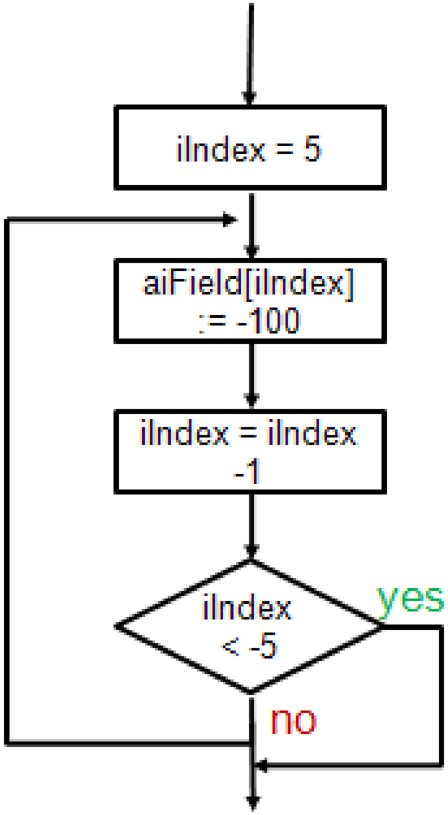
same behaviour

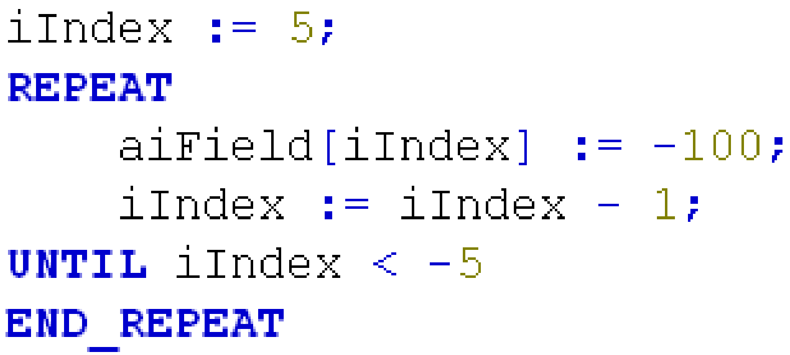
like a FOR loop



Loop (REPEAT)

How to put it into ST?





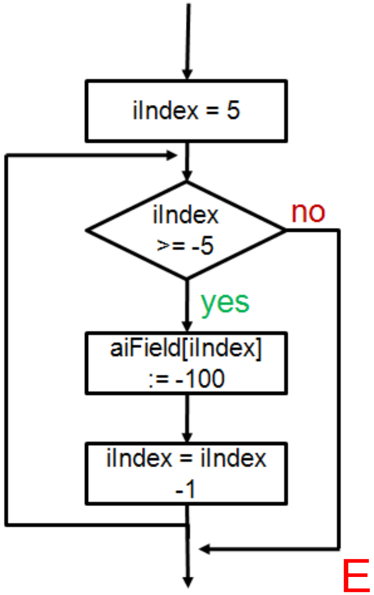
```
iIndex := 5;
REPEAT
    aiField[iIndex] := -100;
    iIndex := iIndex - 1;
UNTIL iIndex < -5
END_REPEAT
```

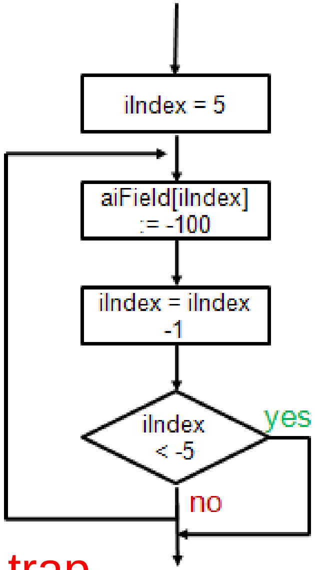
CoDeSys

Structured Text

Language Elements

Loops





⚠

Attention:

Endless loop => Cycle time trap

```
FOR iIndex := 5 TO -5 BY -1 DO
    aiField[iIndex] := -100;
END_FOR
```

```
iIndex := 5;
WHILE iIndex >= -5 DO
    aiField[iIndex] := -100;
    iIndex := iIndex - 1;
END_WHILE
```

```
iIndex := 5;
REPEAT
    aiField[iIndex] := -100;
    iIndex := iIndex - 1;
UNTIL iIndex < -5
END_REPEAT
```

Smart Software Solutions

Further expression

- **EXIT**
  Premature abort of a loop

- **RETURN**
  Premature abort of a POU

- **;**
  Empty expression

Let‘s check

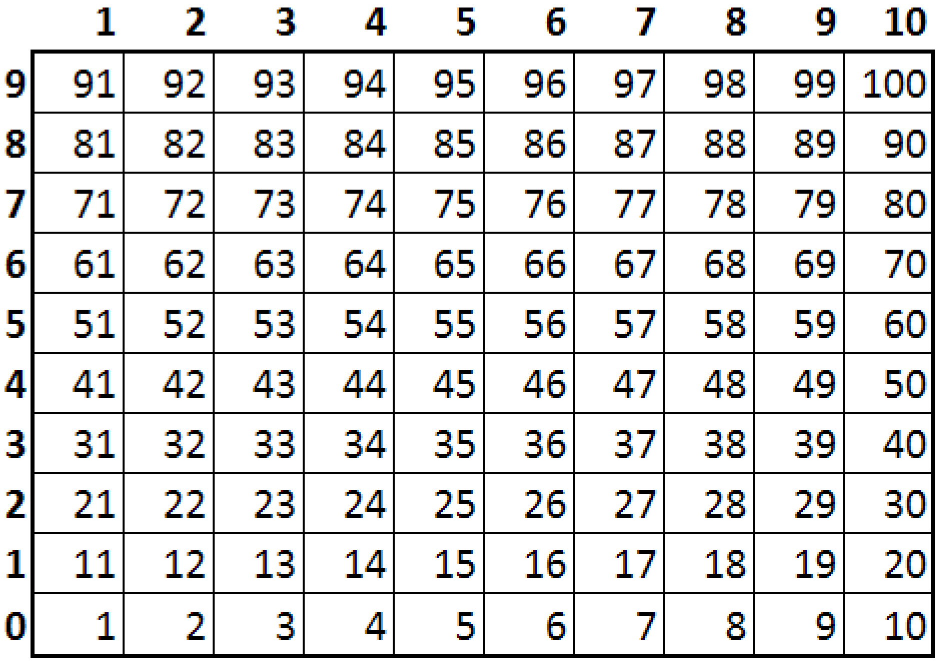
Summary

- ST is a textual language
- The syntax is divided in three different sections…
  - assignments,
  - conditional execution and
  - loops
- It is very powerful to work with big data structures or to initialize some data.

Do some practice

Exercise 1

- Create a variable “aiTable”.
- Initialize “aiTable” upon every PLC cycle with the following pattern.



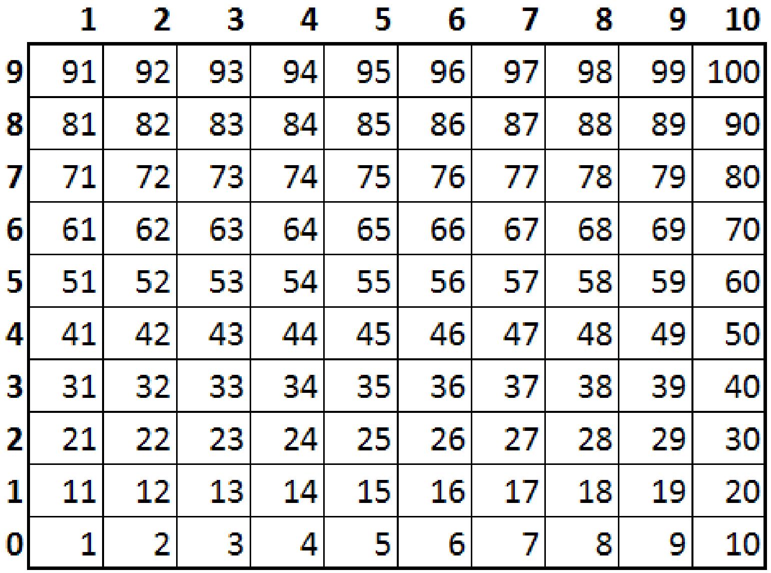
	1	2	3	4	5	6	7	8	9	10
9	91	92	93	94	95	96	97	98	99	100
8	81	82	83	84	85	86	87	88	89	90
7	71	72	73	74	75	76	77	78	79	80
6	61	62	63	64	65	66	67	68	69	70
5	51	52	53	54	55	56	57	58	59	60
4	41	42	43	44	45	46	47	48	49	50
3	31	32	33	34	35	36	37	38	39	40
2	21	22	23	24	25	26	27	28	29	30
1	11	12	13	14	15	16	17	18	19	20
0	1	2	3	4	5	6	7	8	9	10

Do some practice

Exercise 2

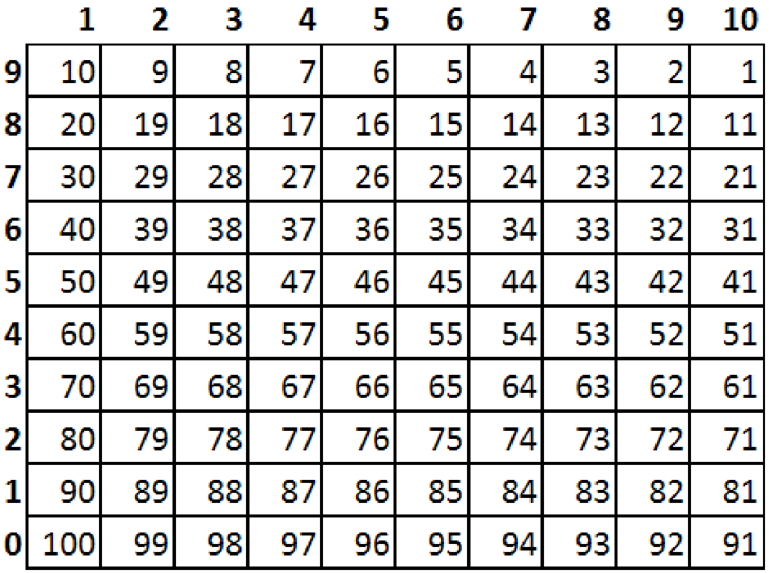
- Extend exercise 1.
- Different init pattern depends on the variable “xDirection”

xDirection = FALSE



	1	2	3	4	5	6	7	8	9	10
9	91	92	93	94	95	96	97	98	99	100
8	81	82	83	84	85	86	87	88	89	90
7	71	72	73	74	75	76	77	78	79	80
6	61	62	63	64	65	66	67	68	69	70
5	51	52	53	54	55	56	57	58	59	60
4	41	42	43	44	45	46	47	48	49	50
3	31	32	33	34	35	36	37	38	39	40
2	21	22	23	24	25	26	27	28	29	30
1	11	12	13	14	15	16	17	18	19	20
0	1	2	3	4	5	6	7	8	9	10

xDirection = TRUE



	1	2	3	4	5	6	7	8	9	10
9	10	9	8	7	6	5	4	3	2	1
8	20	19	18	17	16	15	14	13	12	11
7	30	29	28	27	26	25	24	23	22	21
6	40	39	38	37	36	35	34	33	32	31
5	50	49	48	47	46	45	44	43	42	41
4	60	59	58	57	56	55	54	53	52	51
3	70	69	68	67	66	65	64	63	62	61
2	80	79	78	77	76	75	74	73	72	71
1	90	89	88	87	86	85	84	83	82	81
0	100	99	98	97	96	95	94	93	92	91

Do some practice

Exercise 3

- Extend exercise 2.
- Different init pattern depends on the variable “xDirection”

xDirection = FALSE

	1	2	3	4	5	6	7	8	9	10
9	91	92	93	94	95	96	97	98	99	100
8	81	82	83	84	85	86	87	88	89	90
7	71	72	73	74	75	76	77	78	79	80
6	61	62	63	64	65	66	67	68	69	70
5	51	52	53	54	55	56	57	58	59	60
4	41	42	43	44	45	46	47	48	49	50
3	31	32	33	34	35	36	37	38	39	40
2	21	22	23	24	25	26	27	28	29	30
1	11	12	13	14	15	16	17	18	19	20
0	1	2	3	4	5	6	7	8	9	10

xDirection = TRUE

	1	2	3	4	5	6	7	8	9	10
9	10	9	33	7	33	5	55	55	55	1
8	20	19	33	17	33	15	55	55	55	11
7	30	29	33	27	33	25	55	55	55	21
6	40	39	33	37	33	35	55	55	55	31
5	50	49	33	47	33	45	55	55	55	41
4	60	59	33	57	33	55	55	55	55	51
3	70	69	33	67	33	65	55	55	55	61
2	80	79	33	77	33	75	55	55	55	71
1	90	89	33	87	33	85	55	55	55	81
0	100	99	33	97	33	95	55	55	55	91