1-Programming_Basics_(Structured_Text)_na00189_eng

PLC
Programming Basics (Structured Text)

This course covers how to create basic programs used to control MELSEC programmable controllers.
Structured text (ST) is used for program descriptions in this course.

Copyright ©2015 Mitsubishi Electric Corporation. All Rights Reserved.

Introduction Purpose of the course

This course explains how to create control programs in structured text (ST) for MELSEC programmable controllers.

Completion of the following course or having equivalent knowledge is a prerequisite prior to taking this course:

Programming Basics

Having knowledge or experience with C or BASIC programming language can help understand the contents of this course.

Introduction Course structure

The contents of this course are as follows.

Chapter 1 - Overview of structured text

This chapter describes the features and suitable applications of structured text (ST).

Chapter 2 - Basic rules of ST programs

This chapter describes the basic rules used to create programs in ST.

Chapter 3 - Creating I/O control programs

This chapter describes how to create I/O control programs.

Chapter 4 – Arithmetic operations

This chapter describes how to create arithmetic operation programs.

Chapter 5 - Conditional branching

This chapter describes conditional branching.

Chapter 6 - Storage and handling of data

This chapter describes how to write concise programs to store and handle data.

Chapter 7 – Handling of string data

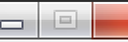
This chapter describes the methods to handle string data.

Final Test

Pass grade: 60% or higher

How to use this e-Learning tool

Go to the next page		Go to the next page.
Back to the previous page		Back to the previous page.
Move to the desired page		"Table of Contents" will be displayed, enabling you to navigate to the desired page.
Exit the learning		Exit the learning.



Cautions for use

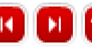
Safety precautions

When you learn based on using actual products, please carefully read the safety precautions in the corresponding manuals.

Precautions in this course

The displayed screens of the MELSOFT engineering software that you use may differ from those in this course.
This course uses the ladder symbols of MELSOFT GX Works3 to create programs.

Chapter 1 Overview of structured text

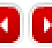


This chapter describes the features and suitable applications of structured text (ST).

1.1 Control programs

1.2 Features of ST and comparison with other IEC programming languages

1.1 Control programs



The following figure illustrates the configuration of a programmable controller system.
Programmable controllers operate according to control programs.
The operation of programmable controllers can be configured as desired by creating control programs.

Input device **Programmable controller** **Output device**

Pushbutton switch
Power
Input interface
Operation part
Memory
Data
Program
Output interface
Lamp
Power
Creating and debugging control programs
PC
Input module
CPU module
Output module
Personal computer

Control program (Ladder)

Pushbutton switch
Lamp
X0
Y10

By writing a control program, the lamp will turn on in response to the status of the pushbutton switch.

The programming languages for programmable controllers are defined by the international standard developed by the International Electrotechnical Commission (IEC).

1.2 Features of ST and comparison with other IEC programming languages

IEC 61131 is an international standard for programmable controller systems.
Programming languages for programmable controllers are standardized by IEC 61131-3. ST is one of the standard programming languages.
Each language has different features to accommodate your application and programmers' skill.

The following table lists features of the IEC 61131-3 programming languages.

Programming language	Features
Ladder **D**iagram (LD)	• Symbols for contacts and coils are used to create a program resembling an electrical circuit. • Program flow is easy to follow and understand, even for beginners.
Structured **T**ext (ST)	• Programs are written in text (characters). • ST is easy to learn for those with experience in writing programs in C or BASIC programming language. • Calculation formulas are similar to mathematical expressions, which are easy to understand. • ST is suitable for data handling.
Function **B**lock **D**iagram (FBD)	• Programs are written by arranging blocks with different functions and indicating relationships between the blocks. • FBD Improves readability as the entire operation can be easily seen.
Sequential **F**unction **C**hart (SFC)	• Conditions and processes are written as flowcharts. • Program flow is easy to understand.
Instruction **L**ist (IL)	• IL is similar to machine language. • IL is rarely used today.

This course describes how to write basic control programs using ST.

1.3 Summary

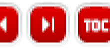
The contents of this chapter are:

- Relationship between programmable controller systems and control programs
- International standard for control programs
- Features of ST

Important points to consider:

Topic	Points
Relationship between programmable controller systems and control programs	• Programmable controllers operate according to control programs. • The operation of programmable controllers can be configured as desired by creating control programs.
International standard for control programs	• ST is one of the IEC programming languages. • Other IEC programming languages include LD, FBD, SFC, and IL, each of which has different features to accommodate your application and programmers' skill.
Features of ST	• ST is easy to learn for those with experience writing programs in C or BASIC language. • Calculations such as addition and subtraction can be written as typically used mathematical expressions, which are easy to understand. • ST is suitable for data handling.

Chapter 2 Basic rules of programs in ST



This chapter describes the basic rules used to create programs in ST.

2.1 Basic program example (I/O control statement)

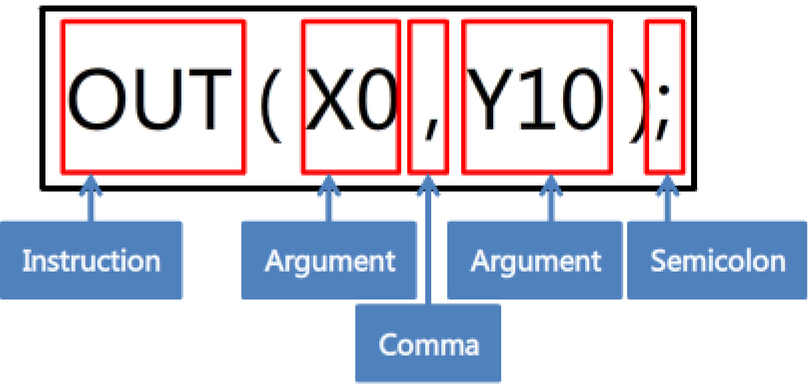
2.2 Basic program example (Assignment statement)

2.3 Numerical notation

2.4 Program execution sequence

2.1 Basic program example (I/O control statement)

This section illustrates an example of a basic ST program.
With the following example program, the output Y0 turns on when the input X10 turns on, and Y0 turns off when X10 turns off.



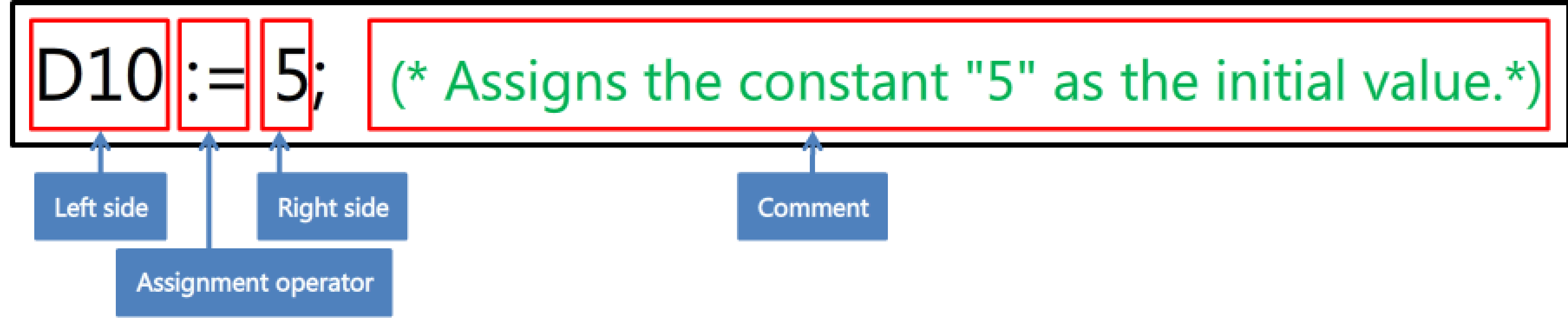
An instruction defines the operation to execute.
Arguments are written in parentheses after an instruction.
Arguments are used to describe variables, arithmetic expressions, and constant values.
With MELSEC programmable controllers, devices of the CPU module can be used as variables.

The number of arguments depends on the instruction.
Multiple arguments are separated with commas (,).

The one line shown above represents one statement. Each statement is ended with a semicolon (;).
A program is written by combining statements.

2.2 Basic program example (Assignment statement)

The next example illustrates a program that uses an assignment statement.
The following statement assigns the decimal constant "5" to the variable "D10".



An assignment operator (:=) is used for this assignment statement. Note that a colon (:) is placed to the left of the equal sign (=).
An assignment operator assigns the right side value to the left side.

Adding a comment to a program makes the operation more understandable. Enclose comments between two asterisks (* *).

2.3 Numerical notation

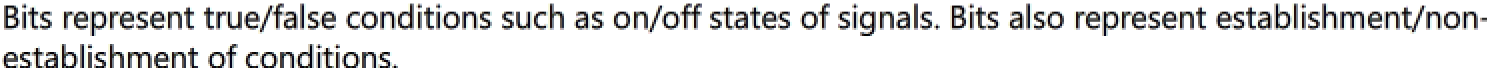
With the example program on the previous page, a decimal value was assigned to a variable.

Sometimes values other than decimal such as binary and hexadecimal are used for sequential control.
The following table lists the types of numerical notation used in ST used for MELSEC programmable controllers.

Type of numerical notation	Notation method	Example
Binary	Add a prefix of "2#".	**2#**11010
Octal	Add a prefix of "8#".	**8#**32
Decimal	Direct input	26
	Add a prefix of "K".	**K**26
Hexadecimal	Add a prefix of "16#".	**16#**1A
	Add a prefix of "H"	**H**1A

Program examples to assign values to variables are shown below.

```
D10 := 8#32;
D10 := K26;
D10 := H1A;
```



2.3.1 Bit notation

Bits represent true/false conditions such as on/off states of signals. Bits also represent establishment/non-establishment of conditions.
In ST, bits cannot be written as "ON" and "OFF". These are expressed as "1" (ON) and "0" (OFF). Bits can also be expressed as "TRUE" and "FALSE".

The following table lists the different types of notation.

State	ON	OFF
	True	False
Numeric notation	1	0
True/false notation	TRUE	FALSE

Here are some examples of assigning values to bit-type variables.

Numeric notation

```
X0 := 1;
```

=

True/false notation

```
X0 := TRUE;
```

Numeric notation

```
X0 := 0;
```

=

True/false notation

```
X0 := FALSE;
```

2.4 Program execution sequence

ST statements are executed in order from top to bottom.

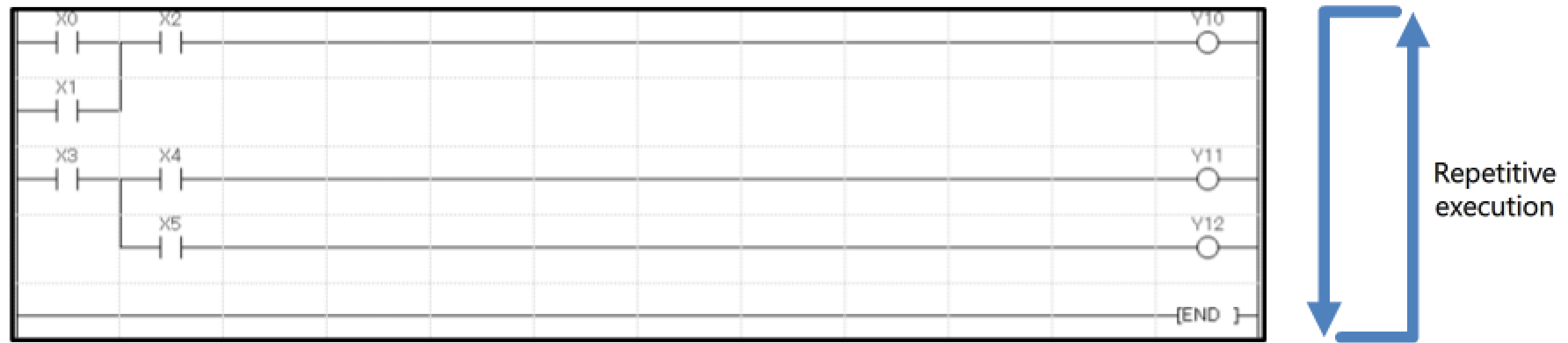
ST program example

```
Y10 := (X0 OR X1) AND X2;          (* Executed first *)
Y11 := X3 AND X4;                  (* Executed second *)
Y12 := X3 AND X5;   (* Executed third. Does not require an END statement at the end.*)
```

Repetitive execution

*Though an END statement is necessary at the end of the program in LD, it is unnecessary in ST.

The following ladder program represents the same operation as the ST program example above.



As is the case in LD, instructions in ST are repeatedly executed by returning to the first instruction after reaching to the last instruction.

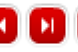
2.5 Summary

The contents of this chapter are:

- Basic ST program
- Assignment statement format
- Numerical notation
- Program execution sequence
- Comment

Important points to consider:

Basic ST program	• A statement is the minimum element of ST programs. • Each statement is ended with a semicolon (;). • A program is written by combining statements.
Assignment statement format	• An assignment operator (:=) is used for assignments.
Numerical notation	• Types of numerical notation in ST • "1" and "0" are used for bit values in ST instead of the "ON" and "OFF" notation. • Bit values can be also stated as "TRUE" and "FALSE" in ST.
Program execution sequence	• Programs created in ST are executed in order from top to bottom. • As with LD programs, ST programs process repetitively returning to the beginning of the program once the end of the process has been reached.
Comment	• Adding a comment to a program makes the operation more understandable. • Comments are enclosed between two asterisks (* *).



Chapter 3 Creating I/O control programs

This chapter describes how to create I/O control programs in ST.

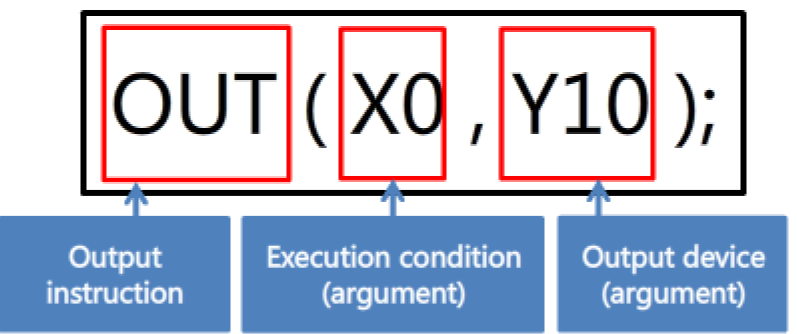
3.1 I/O control programs

3.2 Combining multiple conditions

3.3 Defining meaning to variables

3.1 I/O control programs

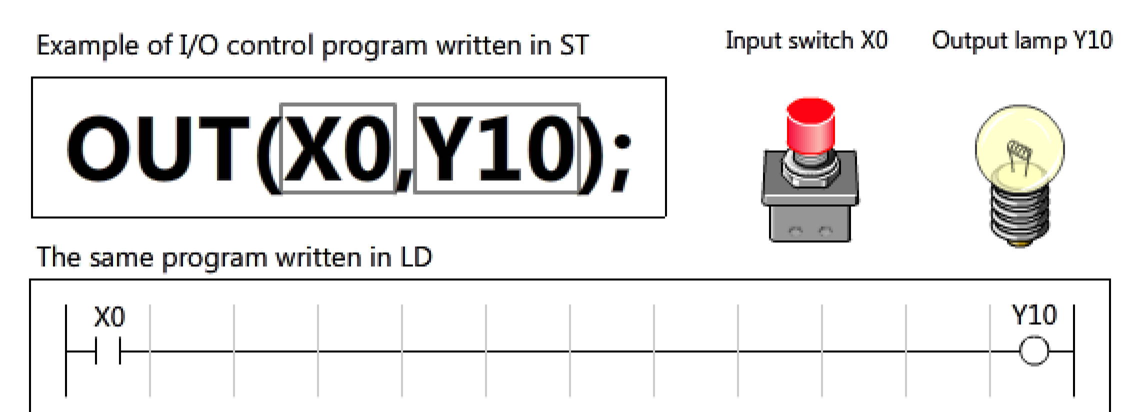
The following is a program example for I/O control of a programmable controller.

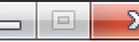


"OUT" is the output instruction. An argument specifies an execution condition and the device to which output is directed. When the execution condition X0 is satisfied, the device Y10 turns on.

Click the input switch shown below. The input switch X0 turns on.

- When the input switch X0 turns on, the output lamp Y10 turns on.
- When the input switch X0 turns off, the output lamp Y10 turns off.





3.1 I/O control programs

Similar to LD, there are many instructions available besides the OUT instruction such as I/O control instructions and data processing instructions.
Refer to the programming manual of your programmable controller for more information on the instructions available in ST.

Note that writing "OUT(X0, Y10);" as "Y10 := X0;" produces the same operation.

```
Y10 := X0;  (* Same operation as "OUT(X0, Y10);" *)
```

3.2 Combining multiple conditions

The following ladder program represents a self-retaining circuit.

The same program can be written in ST as follows.

```
Y70 := (X0 OR Y70) AND NOT X1;
```

Logical operator

As shown above, logical operators are used to combine multiple conditions in ST.

The following table lists the logical operators.

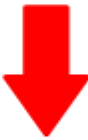
Operator	Meaning
OR	Logical OR
AND	Logical AND
NOT	Logical negation
XOR	Exclusive OR

3.3 Defining meaning to variables

Using ST with MELSEC programmable controllers, both devices and labels can be assigned as aliases to variables.
Users can use labels according to the applications.

When a label related to the application is assigned, the operation is easier to understand.

```
Y10 := (X0 OR X1) AND X2;    (* Written using device names *)
```



```
Lamp := (Switch0 OR Switch1) AND Switch2;    (* Written using labels *)
```

Labels can be named using the MELSOFT engineering software.

Subsequent program examples in this course are described using labels.

3.4 Summary

The contents of this chapter are:

I/O control program examples

- Logical operators are used to combine multiple conditions in ST.
- Device names and labels can be used as variable names.

Important points to consider:

Combining multiple conditions	• Logical operators are used to combine conditions in ST.
Defining meaning to variables	• When a label related to the application is assigned, the operation is easier to understand.

Chapter 4 Arithmetic operations

This chapter describes how to create arithmetic operation programs.

- Describing arithmetic operations
- Specifying data types corresponding to the numeric ranges
- Naming variables to avoid data type inconsistencies

4.1 Basic arithmetic operations

4.2 Data types of variables

4.3 Variable names that represent data types

4.1 Basic arithmetic operations

This example program totals the production volume of two separate production lines.
The right side of an equation is an arithmetic operation containing variables and arithmetic operators.

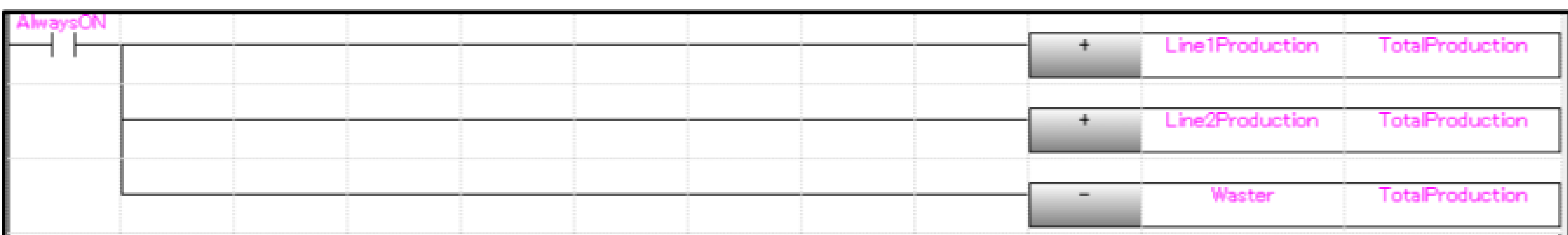
Example arithmetic program written in ST

Addition operator

Subtraction operator

```
TotalProduction := Line1Production + Line2Production - Waster;
(* Total the production volume of two production lines, subtract the number of
defective products from the total, and assign the obtained value. *)
```

The same program written in LD is as shown below.

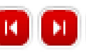


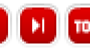
As shown above, the program must be written by using 3 lines in Ladder, but with ST, it can be written in 1 line.

The following table lists the basic arithmetic operators.

Operator	Meaning
+	Addition
-	Subtraction
*	Multiplication
/	Division

4.2 Data types of variables





A data type must be specified for each variable to define the range of the values to be handled.
The data types for numeric values used in ST are bit, integer, and real number types.

Among the data types used in ST, the table below shows the data types used in this course.

Data type		Data range
Bit		ON/OFF state of bit devices and true/false state of execution results
Integer	Word (unsigned)	0 - 65,535
	Word (signed)	-32,768 - 32,767
	Double-word (unsigned)	0 - 4,294,967,295
	Double-word (signed)	-2,147,483,648 - 2,147,483,647

When using the integer type, select the word or double-word type according to the data range, and select the signed or unsigned type according to the necessity to handle negative values.
Specify the data type of a variable when the label name is set using the MELSOFT engineering software.

4.3 Variable names that represent data types

Using different data types on the left and right sides of an assignment equation may cause a compiling error or unexpected result. Below is an example of such a case.

```
ValueA := ValueB;     (* ValueA: Word integer  ValueB: Double-word integer *)
```

A double-word integer cannot be assigned to a word integer. However in this case, the data type is not recognizable.

Prefixes that represent the data type can be added to variable names to make data types to be visually identifiable. This kind of variable naming is known as Hungarian notation.

Data type		Data range	Prefix	Expansion of prefix
Bit		ON/OFF state of bit devices and true/false state of execution results	b	**B**it
Integer	Word (unsigned)	0 - 65,535	u	**u**nsigned word
	Word (signed)	-32,768 - 32,767	w	signed **w**ord
	Double-word (unsigned)	0 - 4,294,967,295	ud	**u**nsigned **d**ouble-word
	Double-word (signed)	-2,147,483,648 - 2,147,483,647	d	signed **d**ouble-word

The example program on top of this page can be written as follows using Hungarian notation:

```
wValueA := dValueB;  (* Double-word variable cannot be assigned to a word variable. *)
```

By using Hungarian notation, data type inconsistencies can be identified in the process of writing a program.

In the rest of the course, variable names in example are written in Hungarian notation.

4.4 Summary

The contents of this chapter are:

- Describing arithmetic operations
- Specifying data types corresponding to the numeric ranges
- Adding variable names that represent data types

Important points to consider:

Basic arithmetic operations	• Operators common to general programming languages can be used in ST to express calculations.
Data types of variables	• A data type must be specified for each variable to define the range of the values to be handled.
Adding variable names that represent data types	• Describing variable names using Hungarian notation enables inconsistencies in variable data types to be identified when writing programs.

Chapter 5 Conditional branching

Control programs also contain sections of code in which actual processing changes in accordance with specified conditions. This chapter describes conditional branching.

5.1 Conditional branching (IF)

5.2 Conditional branching according to integer values (CASE)

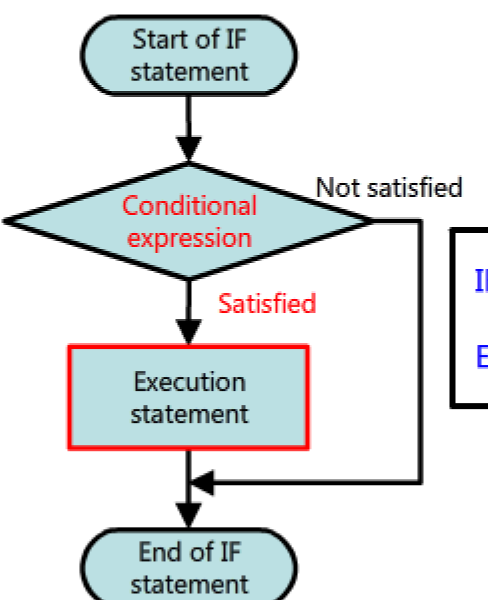
5.1 Conditional branching (IF)

IF statements are used for conditional branching. IF statements are described as follows.

```
IF conditional expression THEN
  Execution statement;          (* Statement is executed if conditional expression is satisfied. *)
END_IF;                         (* END_IF; must be placed at the end of the IF statements. *)
```

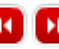
In this example program, a statement is executed when the conditional expression is satisfied. The statement is not executed when the conditional expression is not satisfied.

The following figure illustrates the operation flow in this example program.



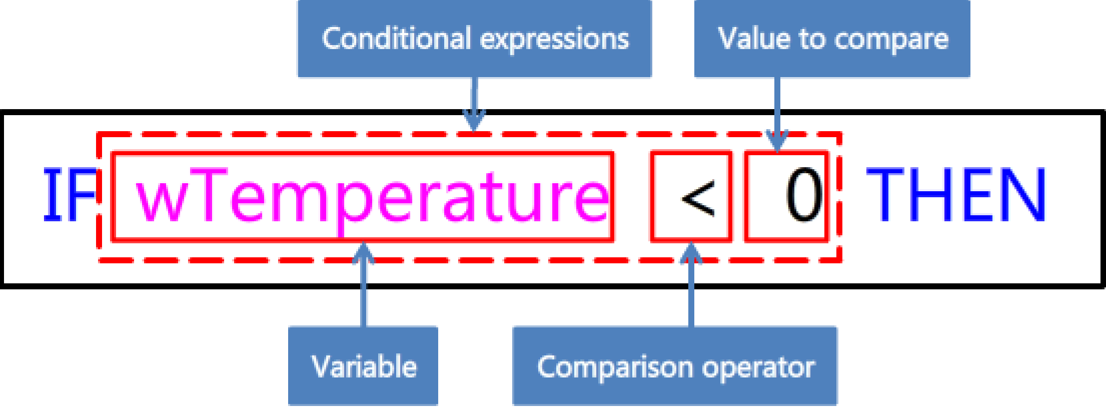
The following example illustrates branching of the program by comparing values of variables. In the example program, the heater turns on when the temperature in the control panel falls below 0 degrees.

```
IF wTemperature < 0 THEN
 bHeater := 1;  (* The heater is turned on when the temperature in the control panel falls below 0 degrees. *)
END_IF;
```



5.1.1 Writing conditional expressions

The previous page described a conditional expression of "wTemperature < 0", which means "when the value of variable wTemperature is less than 0".
Like this expression, conditional expressions use comparison operators to represent relationship between variables and values to compare.



On the left and right sides of an comparison operator, values are written as variables or constants for comparison.

In addition to comparing variables and constants, conditional expressions can be written to compare variables, and perform logical operations of comparison results or bit-type variables.

Comparing variables

- uValue1 <= uValue2

Logical operation for two comparison results

- (10 < uValue) AND (uValue <= 50)

Logical operation for two bit-type variables

- bSwitch0 OR bSwitch1

The following table lists the types of comparison operators.

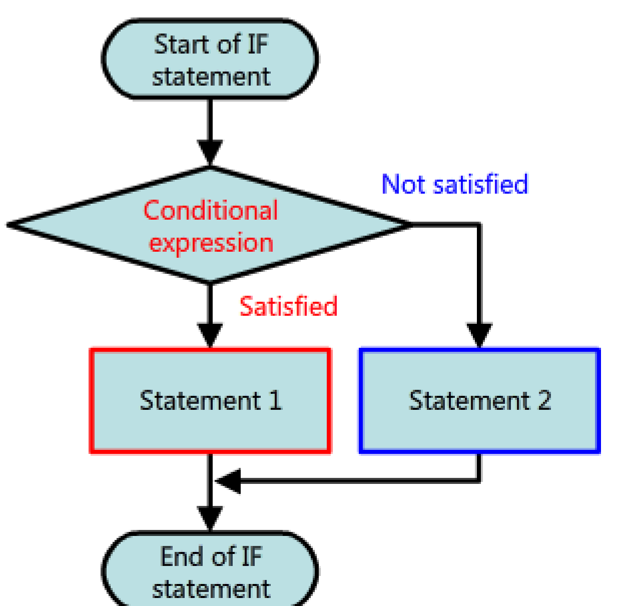
Operator	Meaning
>	Greater than
<	Less than
>=	Greater than or equal to
<=	Less than or equal to
=	Equals
<>	Does not equal

5.1.2 Exceptional branching for IF statement (ELSE)

Simple IF statements (see 5.1) are used to execute a statement when the conditional expression is satisfied.
To execute a different statement when the conditional expression is not satisfied, an ELSE statement is used.

```
IF conditional expression THEN
  Execution statement 1;          (* Statement 1 is executed if conditional expression is satisfied. *)
  ELSE
  Execution statement 2;          (* Statement 2 is executed if conditional expression is not satisfied *)
END_IF;
```

The following figure illustrates the operation flow when using an ELSE statement.



The following example program executes different statements depending on whether the condition is satisfied.

The example program in 5.1 had a drawback that the heater keeps raising the temperature even after it reaches 0 degrees. However, the following program turns off the heater when "wTemperature" exceeds 0 degrees.

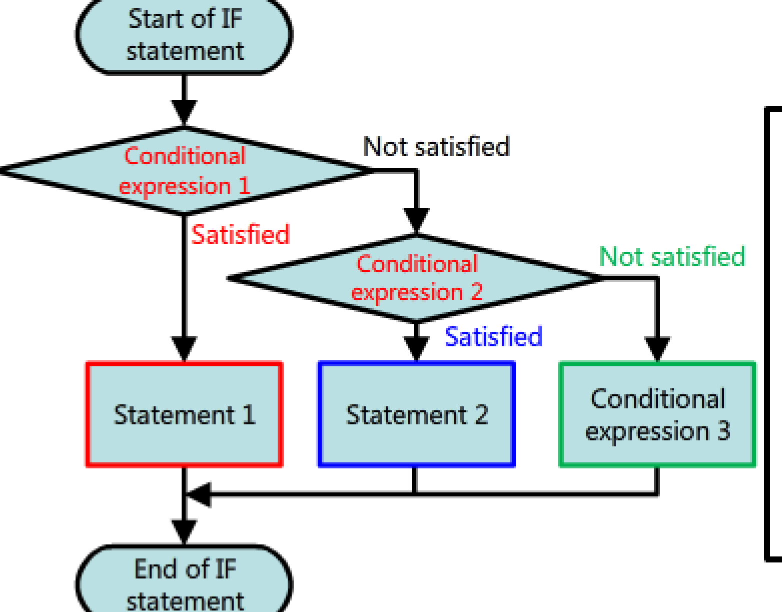
```
IF wTemperature < 0 THEN
  bHeater := 1;  (* Turns on the heater when the temperature falls below 0 degrees. *)
  ELSE
  bHeater := 0;  (* Turns off the heater when the temperature reaches or exceeds 0 degrees *)
END_IF;
```

5.1.3 Additional branching for IF statement (ELSIF)

ELSE statements are used to execute a different statement when the conditional expression is not satisfied.
Another conditional branching can be added by using ELSIF statements, which mean that if the previous conditional expression is not satisfied, then another conditional expression is checked.

```
IF Conditional expression 1 THEN
  Execution statement 1;      (* Statement 1 is executed if conditional expression 1 is satisfied. *)
  ELSIF Conditional expression 2 THEN
  Execution statement 2;      (* Statement 2 is executed if conditional expression 1 is not satisfied and conditional expression 2 is satisfied. *)
  ELSE
  Execution statement 3;      (* Statement 3 is executed if conditional expressions 1 and 2 are not satisfied. *)
END_IF;
```

The following figure illustrates the operation flow when using an ELSEIF statement.



ELSIF statement is added to the program example illustrated in 5.1.2 to cope with the case when the temperature exceeds 40 degrees.

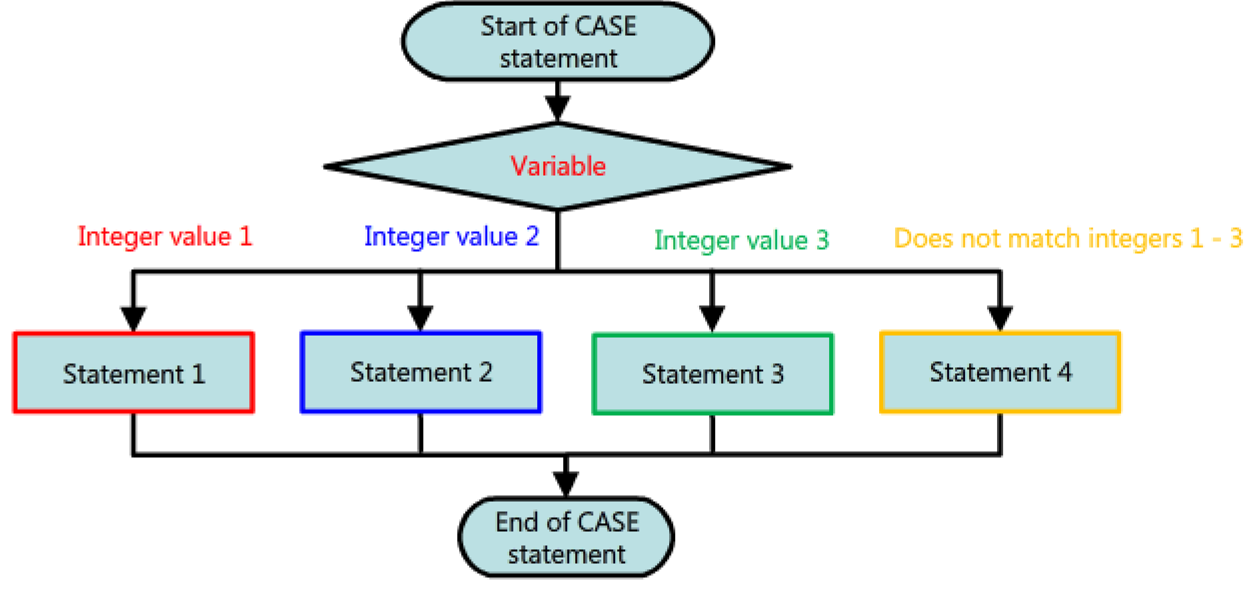
```
IF wTemperature < 0 THEN
  bHeater := 1; (* Turns on the heater when the temperature falls below 0 degrees. *)
  bCooler := 0; (*  Turns off the cooler when the temperature falls below 0 degrees. *)
  ELSIF 40 < wTemperature THEN
  bHeater := 0; (* Turns off the heater if the temperature exceeds 40 degrees. *)
  bCooler := 1; (* Turns on the cooler if the temperature exceeds 40 degrees. *)
  ELSE
  bHeater := 0; (*  Turns off the heater if none of the previous conditions are satisfied. *)
  bCooler := 0; (*  Turns off the cooler if none of the previous conditions are satisfied. *)
END_IF;
```

5.2 Conditional branching according to integer values (CASE)

IF statements are used for branching depending on whether or not conditional expressions are satisfied.
CASE statements are used for branching according to integer values.
The following figure illustrates how a CASE statement is written.

```
CASE Variable OF
  Integer value 1: Execution statement 1;    (* Statement 1 is executed when the variable matches integer value 1. *)
  Integer value 2: Execution statement 2;    (* Statement 2 is executed when the variable matches integer value 2. *)
  Integer value 3: Execution statement 3;    (* Statement 3 is executed when the variable matches integer value 3. *)
  ELSE Execution statement 4;    (* Statement 4 is executed if the variable does not match any of the integer values. *)
END_CASE;    (* "END_CASE;" must be placed at the end of the CASE statement. *)
```

The following figure illustrates the operation flow when using a CASE statement.



5.2.1 Example program for CASE statement

The CASE statement execution is described using the example program operation.

Click [▶] to proceed to the next page.
To watch the animation again, click the "Play" button.

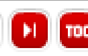
Play

Weight:
Grade:

```
CASE wWeight OF
 0..20:     uSize := 1;
 21..30:    uSize := 2;
 31..40:    uSize := 3;
 ELSE       uSize := 4;
END_CASE;
```

Weight	uSize	Grade
0 to 20 kg	1	M
21 to 30 kg	2	L
31 to 40 kg	3	XL
41 kg and over	4	Oversize

5.3 Summary

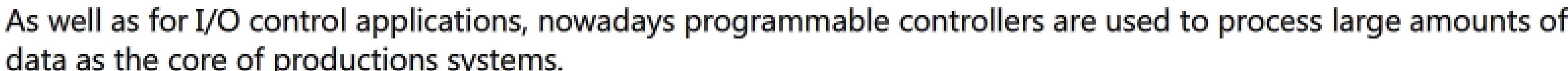


The contents of this chapter are:

- Conditional branching with IF statements
- Writing conditional expressions
- Conditional branching according to integer values (CASE statement)

Important points to consider:

IF statement	• With an IF statement, the program is branched when a conditional expression is satisfied. • An ELSE statement is used for branching when the conditional expression is not satisfied. • An ELSIF statement is used to add another branching when the conditional expression in the IF statement is not satisfied.
Conditional expression	• Conditional expressions represent relationship between variables and values to compare using comparison operators.
CASE statement	• CASE statements are used for branching according to integer values.



As well as for I/O control applications, nowadays programmable controllers are used to process large amounts of data as the core of productions systems.
To process large amounts of data, data must be stored and then read as necessary.
This chapter describes how to write concise programs to store and process data.

- Arrays are used to sequence and organize variables.
- Data structures are used to organize related variables.
- Loop-processing programs effectively process arrays using FOR statements.

Concise programs to store and handle data can be created by using arrays, data structures, and FOR statements.

6.1 Sequencing and storing data (Array)

6.2 Looping (FOR)

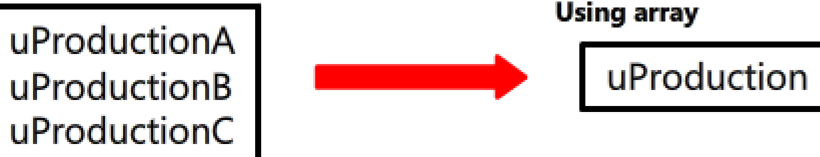
6.3 Storing related data (Structures)

6.1 Sequencing and storing data (Array)

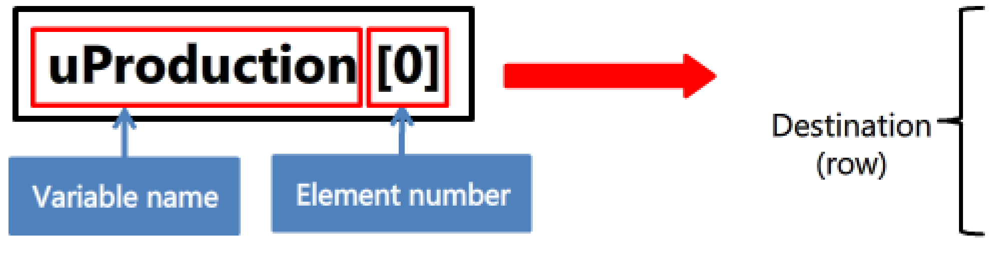
By using arrays, multiple values can be handled by one variable.
In the following example, the production volume data in an automotive manufacturing plant is stored by destination country.

Destination	Country A	Country B	Country C
Production volume	35	75	65

The production volume data by destination country is assigned to a variable. Without using arrays, one variable must be created for each destination.
By using arrays, however, the production volume for multiple destinations can be assigned to and stored in one variable.



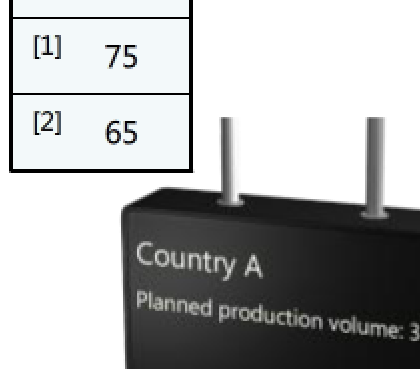
Individual variables in the array are specified using element numbers. Element numbers start from zero [0].



Destination (row)

Country A	[0]	35
Country B	[1]	75
Country C	[2]	65

In the following program example, the variable of the planned production volume for Country A is assigned.




```
uShowProductionPlan := uProduction[0];
(* Specifies the element number for country A. *)
```

6.1.1 Matrix array

Next, paint color data is used in addition to the destination data.

Destination	Country A			Country B			Country C		
Paint color									
Production volume	10	5	20	15	40	20	25	30	10
	35 in total			75 in total			65 in total		

As illustrated in the following table, data can be separated and stored by paint color (column) for each destination country (row).

Paint color (column)

Destination (row)	Red	Yellow	Blue
Country A	[0,0] 10	[0,1] 5	[0,2] 20
Country B	[1,0] 15	[1,1] 40	[1,2] 20
Country C	[2,0] 25	[2,1] 30	[2,2] 10

Element number representing the destination

Element number representing the paint color

Array variable (matrix array)

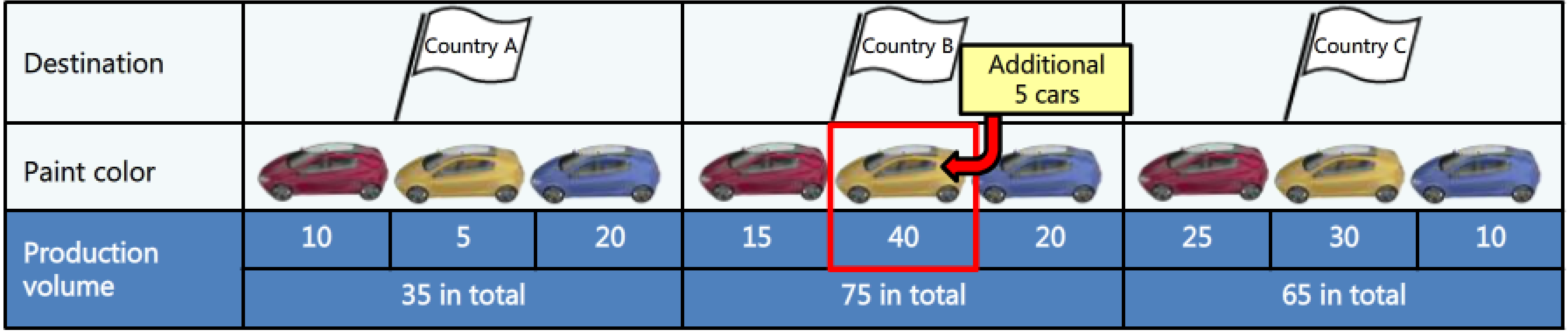
```
uProduction[1,1]
```

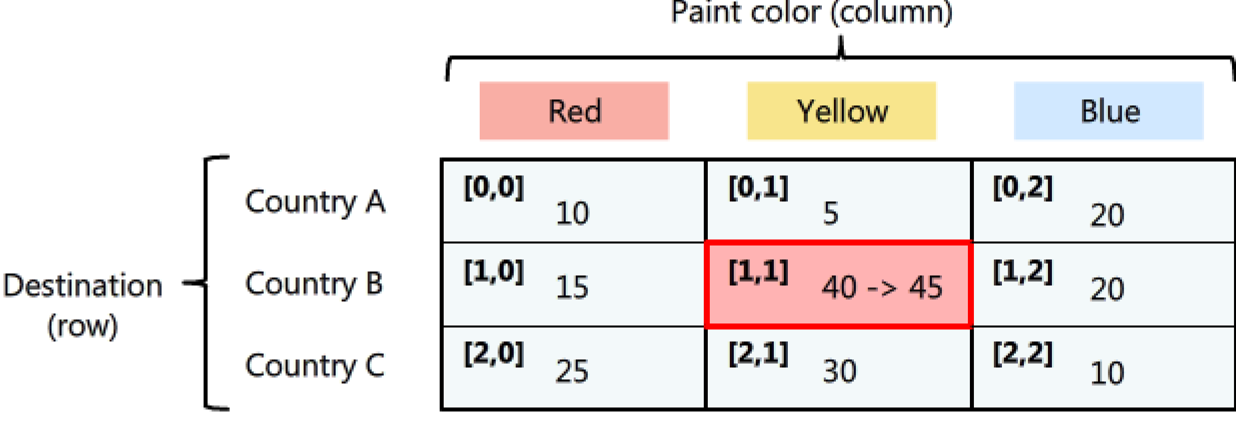
Arrays that organize data into rows and columns in this way are known as matrix arrays.
Element numbers that represent the rows and columns are separated by commas.

6.1.2 Matrix array assignment

Using matrix arrays, the following example program assigns the number of cars to be urgently manufactured in addition to the planned production volume of yellow automobiles for Country B.

```
uAdditionalProduction := 5;
uProduction[1,1] := uProduction[1,1] + uAdditionalProduction;
(* Adds the additional amount of production (5 units) to the initially planned production volume. *)
```

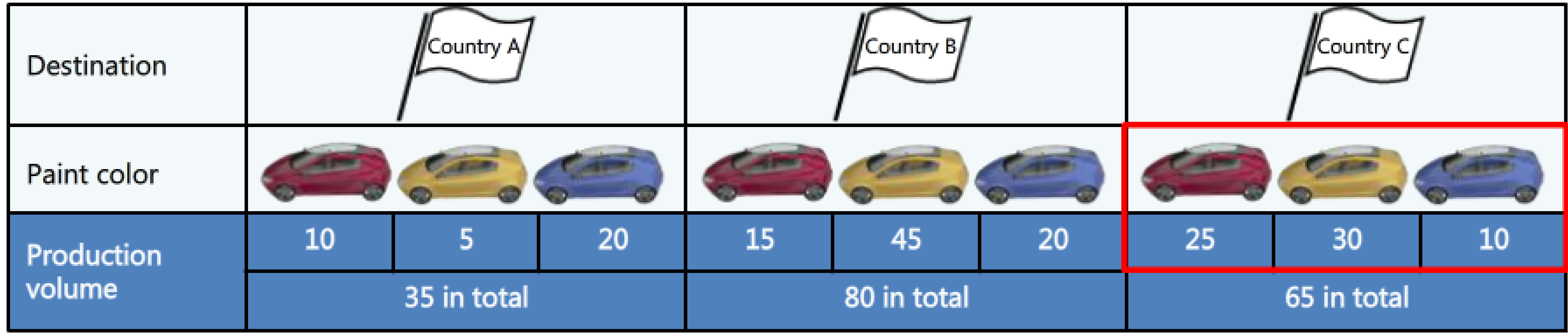




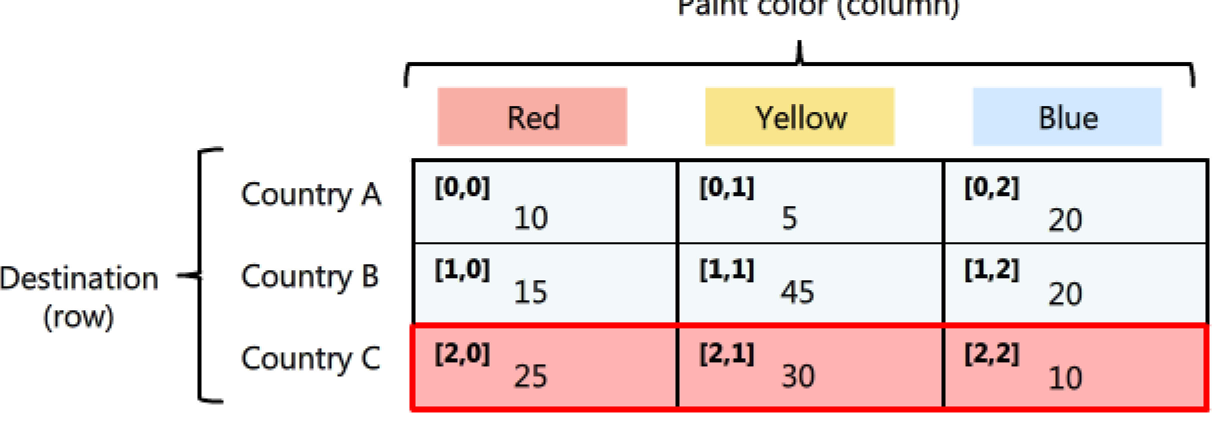
6.1.3 Processing information stored in matrix arrays

Using matrix arrays, the following example program calculates the total production volume planned for all paint colors for Country C and assigns the value to a variable.

```
uProductionToday := uProduction[2,0] + uProduction[2,1] +uProduction[2,2];
(* Calculates the total production volume planned for today for all paint colors for Country C and assigns the value to "uProductionToday". *)
```



Destination	Country A			Country B			Country C		
Paint color									
Production volume	10	5	20	15	45	20	25	30	10
	35 in total			80 in total			65 in total		



Paint color (column)

Destination (row)	Red	Yellow	Blue
Country A	[0,0] 10	[0,1] 5	[0,2] 20
Country B	[1,0] 15	[1,1] 45	[1,2] 20
Country C	[2,0] 25	[2,1] 30	[2,2] 10

6.2 Looping (FOR)

The example program on the previous page (the planned production volume for today is assigned) is shown below again.

```
uProductionToday := uProduction[2,0] + uProduction[2,1] + uProduction[2,2];
```

With this program example, when the number of paint colors increases, more variables will be added. Then, the expression becomes longer, making it more difficult to read.

```
uProductionToday := uProduction[2,0] + uProduction[2,1] + uProduction[2,2]
                    + uProduction[2,3] + uProduction[2,4] + uProduction[2,5] ...
```

In this case, loop statements can be used to create cleaner code.

Loop statements include the FOR, WHILE, and REPEAT statements. This course covers FOR statements.

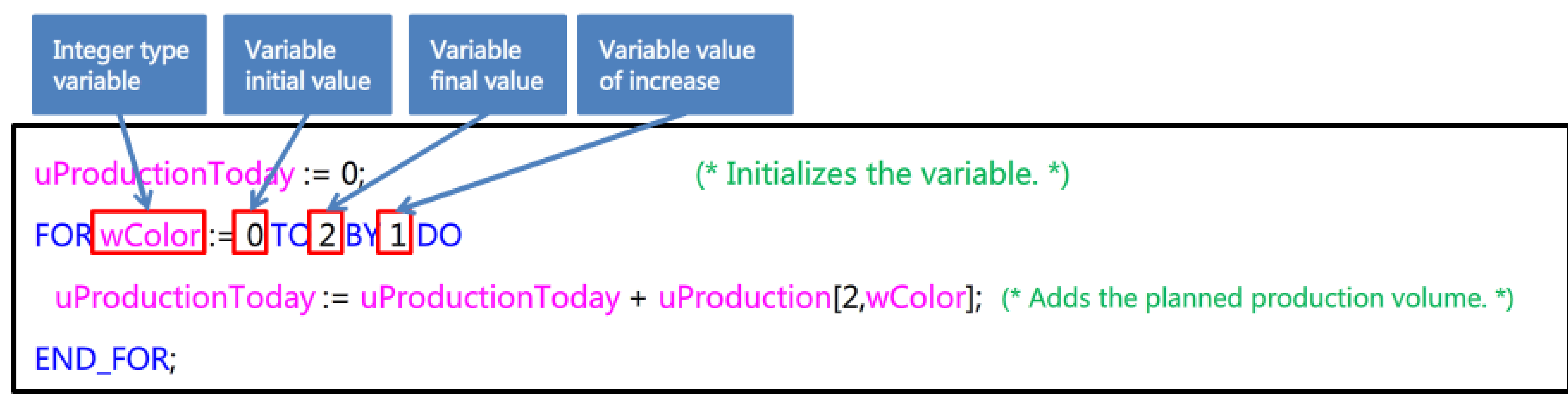
FOR statements are described as follows.

```
FOR variable := initial value TO final value BY increments DO
   Execution statement; (* Statement is executed in a loop until the variable reaches the final value. *)
END_FOR;               (* END_FOR; must be placed at the end of the FOR statements.  *)
```

The statement is repeated until the final value of the variable is reached and the "END_FOR;" code is executed.

6.2 Looping (FOR)

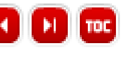
Using a FOR statement, the following example program obtains the planned production volume for all paint colors for Country C.



Using the FOR statement, the "wColor" variable increases by one starting from the initial value of zero and the statement is repeated until the variable reaches the final value of two.
The "wColor" variable is specified as the second element number in the "uProduction" array described in the execution statement.
The value of the "wColor" variable increases each time the statement is repeated. The planned production volume for each paint color is added each time to obtain the total.

This example program is executed in a loop three times. (First time: red [0] => Second time: yellow [1] => Third time: blue [2])

The operation of this program is illustrated on the next page.

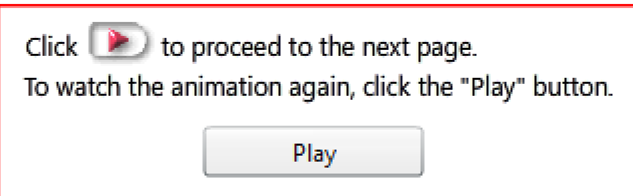


6.2 Looping (FOR)

The execution of the FOR statement is described using the operation of the program example.

Array of estimated production volume

	Red	Yellow	Blue
Country A	[0,0] 10	[0,1] 5	[0,2] 20
Country B	[1,0] 15	[1,1] 45	[1,2] 20
Country C	[2,0] 25	[2,1] 30	[2,2] 10



Number of repetition of the loop: 3

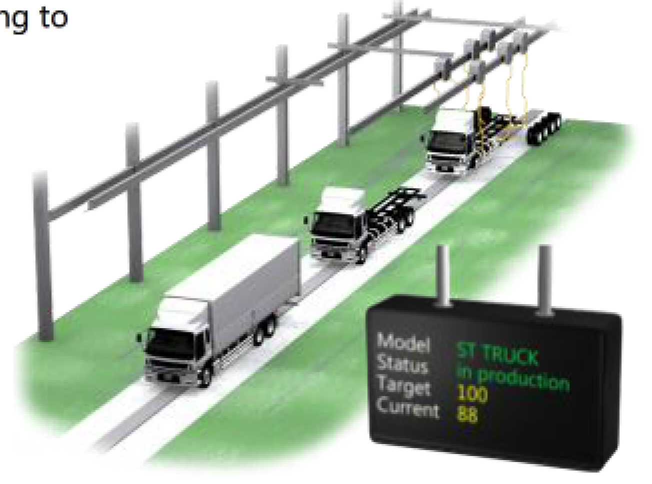
```
uProductionToday := 0;

FOR wColor := 0 TO 2 BY 1 DO
            2
    uProductionToday := uProductionToday + uProduction[2,wColor];
                                   65
END_FOR;
```

6.3 Storing related data (Structure)

A structure enables one variable name to represent multiple related variables.
In the following example, the status of an automobile production line is displayed on Andon (display board).

The following table lists the variable names, values, and the data types corresponding to the displayed items.

Item	Variable name	Value	Variable Data Type
Model	sModel	'ST TRUCK'	Text string
Status	bStatus	'in production'	Bit type
Target production volume for today	uPlanQty	'100'	Integer type Word (unsigned)
Current production volume	uActualQty	'88'	Integer type Word (unsigned)



If a structure is not used, the variable names must be changed for each line when multiple production lines exist.
The following shows examples of variable names by the production line.

First production line

s1stLineModel
b1stLineStatus
u1stLinePlanQty
u1stLineActualQty

Second production line

s2ndLineModel
b2ndLineStatus
u2ndLinePlanQty
u2ndLineActualQty

When the number of production line increases, the number of variables to be handled will also increase. Then, the program becomes longer and more difficult to read.

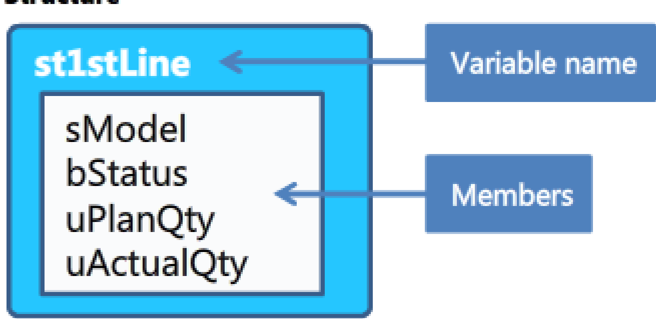
6.3 Storing related data (Structure)

Using structures enables one variable name to represent multiple variables related to one production line.
Like this, structures are used to organize, store, and handle data in a batch for conditions and specifications of physical objects such as devices, equipment and workpieces.

Multiple variables

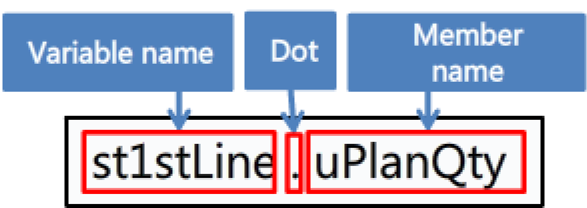
s1stLineModel
b1stLineStatus
u1stLinePlanQty
u1stLineActualQty

Multiple variables defined in a structure



The **st**ructure variable contains a prefix of "st" to represent that this is a structure.
The individual variables defined by the structure are known as members. Data types of each member can be different.

Each member of structure arrays can be specified after the element number of the array using a dot before the member name.



In the following example program, a constant is assigned to a member of the structure variable for the first production line.

```
st1stLine.uPlanQty := 150;
(* Sets the today's target production of the first production line to 150. *)
```

6.3.1 Storing structure arrays

Structures can be created as arrays.
In the following example, the production status is stored by date.

Structure arranged* by date (stStructureByDate)

* In this array, the element number starts from "1".

Day (column)

Month (row)	Day 1			Day 21	
January	[1,1]	[1,2]	...	[1,21]	...
	[2,1]	...	...	...	...
	...	...	...	...	...
	...	...	...	...	...
July	[7,1]	...	...	[7,21]	...
	...	...	...	...	...


```
stProductionByDate[7,21] := st1stLine;
 (* The production status on July 21 is stored in the structure arranged by
date (stProductionByDate). *)
```

Like this, members do not need to be individually specified for structure assignment.

6.3.2 Reading structure arrays

In the following example, the production volume is read from a structure arranged by date and then assigned to a variable.

Structure arranged by date (stProductionByDate)

Day (column)

Day 1

Month (row)

January	[1,1]	[1,2]	...	...	...
	[2,1]	...	...	...	...
	...	...	...	...	...
June	[6,1]	...	...	...	...
	...	...	...	...	...
	...	...	...	...	...

Structure to which the production status on June 1 is stored

stProductionByDate[6,1]

sModel
bStatus
uPlanQty
uActualQty

Assignment

Variable to which the production volume is assigned

uPastProduction

```
uPastProduction := stProductionByDate[6,1].uActualQty;
(* Assigns the production volume on June 1 to the uPastProduction variable. *)
```

Each member of the structure arrays can be specified by appending a dot (.) and a member name to the element number of the array.

6.4 Summary

The contents of this chapter are:

- Overview and usage of arrays
- Loop processing using FOR statements
- Overview and use of structures

Important points to consider:

Array	• Multiple values can be handled by one variable by using arrays. • Individual variables in arrays are specified by element numbers added to the end of variable names.
FOR statement	• Loop statements are used in programs when repetitive operation is desired. • FOR statements are used to repeat operation until the conditions for the end of the loop operation are satisfied. The statement before the "END_FOR;" statement are executed repeatedly.
Structure	• Structures enable one variable name to represent multiple related variables. Structures can include variables of different data types. • Individual variables, or members, defined in structures are specified by adding a dot and the member name after the structure variable name.

Chapter 7 Handling of string data

In some cases, programmable controllers use string data to send commands to or receive feedback from connected devices such as barcode readers, temperature controllers, or electronic scales. For such purposes, it is necessary to join or extract string data as required.

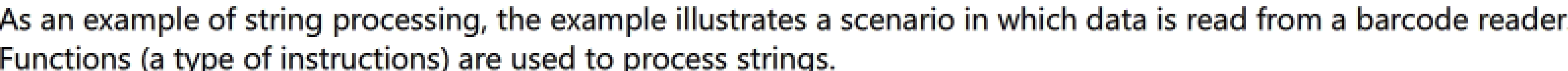
This chapter describes how to handle string data.

7.1 Example of string data handling

7.2 String assignment

7.3 Extracting strings (LEFT)

7.4 Extracting strings (MID)



7.1 Example of string data handling

As an example of string processing, the example illustrates a scenario in which data is read from a barcode reader.
Functions (a type of instructions) are used to process strings.

As illustrated below, strings read by the barcode reader contain a 4-character fixed-length error code and 8-character fixed-length month, date, time and minute data.
The string processing program example will be described using this system.

Example string data read from a barcode reader

e112, 12091458

4-character error code

8-character error generation date

String processing functions

Programmable controller

Barcode reader

Barcode

1 560025 487204

e112

12091458

An error code is extracted.
7.3 Extracting strings (LEFT)

The error occurrence date and time (14:58, December 9) is extracted.
7.4 Extracting strings (MID)

7.2 String assignment

Before explaining about how to extract strings, this section describes the data types for strings.

The data types for strings that can be used with programmable controllers are listed in the following table.

Data type	Character type can be processed	Hungarian notation prefixes	Expansion of prefix
String	Strings of alphanumeric characters and numbers (ASCII) or Japanese (Shift-JIS)	s	**s**tring
String [Unicode]	Strings of many different languages and symbols	ws	**w**ide **s**tring

The type of string to be used depends on the device being connected to the programmable controller or the corresponding language.
This chapter describes different types of text strings.

When a string type string is assigned to a string variable, enclose the string in single-quotation marks (').

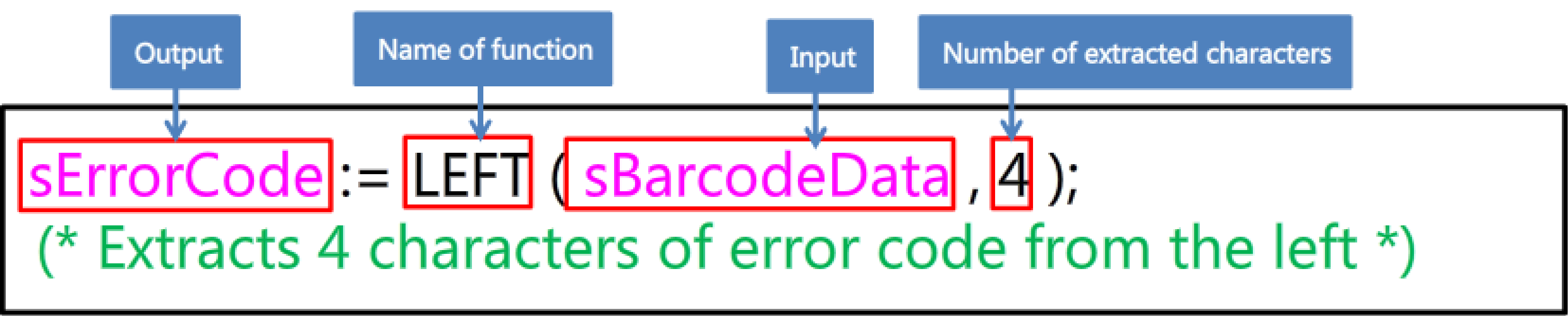
```
sDefault := 'e112,12091458';  (* String assignment *)
```

7.3 Extracting strings (LEFT)

The error code "e112" is extracted from the string variable "sBarcodeData" that contains the string "e112,12091458".

Variable name	Stored string
sBarcodeData	e112, 12091458

The LEFT function extracts only the specified number of characters starting from the left side of the input string.
The following illustrates an example program.



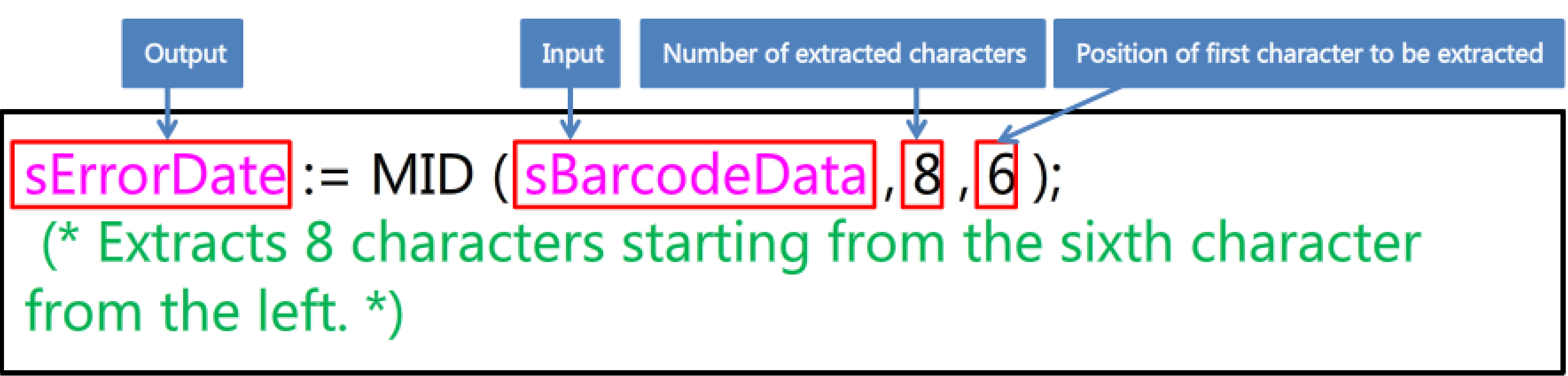
Four characters are extracted from the left. A value "e112", which is the string representing the error code, is assigned to the left side.

7.4 Extracting strings (MID)

The error generation time "12091458" is extracted from the string variable "sBarcodeData" that contains the string "e112,12091458".

Variable name	Stored string
sBarcodeData	e112,12091458

The MID function extracts the specified number of characters from the specified start position in the input string.
The following illustrates an example program.



In this example, an 8-character string is extracted starting from the sixth character. A value "12091458", which is the string representing the error occurrence time, is assigned to the left side.

7.5 Summary

The contents of this chapter are:

- Methods of assigning strings to string variables
- Functions that extract strings (LEFT and MID)

Important points to consider:

String assignment	• To assign a string to a string variable, enclose the string in single-quotation marks ('). • Use either of the string type or the string [Unicode] type according to the device being connected to the programmable controller or the corresponding language.
Functions for handling strings	• Functions are used to handle strings.

7.6 Course summary

This course covered the basics on how to create programs in ST.
This brings us to the end of this e-learning course.

ST programs are created using the MELSOFT engineering software.
For the details of the specific steps such as entering data, editing, saving, and compiling programs with the MELSOFT engineering software, refer to the following.

- Mitsubishi FA e-Learning Course "MELSOFT GX Works3 (Structured Text)" (to be released soon)
- Operating manual of your MELSOFT engineering software

For further information on ST, refer to the following.

- Programming guidebook of your programmable controller

For the information on instructions and functions for your application, refer to the following.

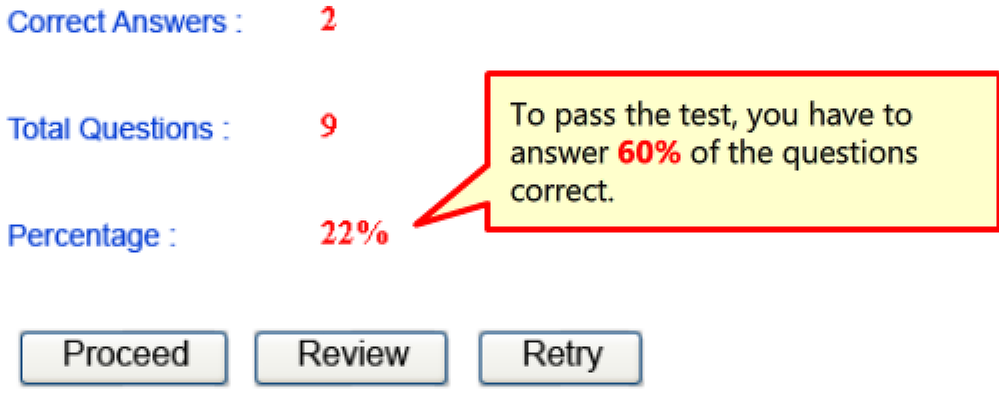
- Programming manual of your programmable controller

Test Final Test

Now that you have completed all of the lessons of the Programming Basics (Structured Text) course, you are ready to take the final test. If you are unclear on any of the topics covered, please take this opportunity to review those topics.
There are a total of 12 questions (20 items) in this Final Test.
You can take the final test as many times as you like.

How to score the test
After selecting the answer, make sure to click the **Answer** button. Your answer will be lost if you proceed without clicking the Answer button. (Regarded as unanswered question.)

Score results
The number of correct answers, the number of questions, the percentage of correct answers, and the pass/fail result will appear on the score page.



- Click the **Proceed** button to exit the test.
- Click the **Review** button to review the test. (Correct answer check)
- Click the **Retry** button to retake the test again.

Test

Final Test 1

Characteristics of structured text (ST)
Select the incorrect description of ST.

- ST is easy to learn for those with experience writing programs in C or BASIC language.
- Calculations such as addition and subtraction can be written as typically used mathematical expressions.
- Symbols for contacts and coils are used to create a program resembling an electrical circuit.
- ST is suitable for data handling.

Answer Back

Test Final Test 2

Basic principles of ST
Select the correct statement written in ST.

- ○ uProduction = 15
- ○ uProduction := 15:
- ● uProduction := 15;
- ○ uProduction = 15;

Answer Back

Test Final Test 3

Describing comments
Select the correct comment written in ST.

- ' Assigns a value of 1 to the variable.
- (* Assigns a value of 1 to the variable. *)
- { Assigns a value of 1 to the variable. }
- <!-- Assigns a value of 1 to the variable. -->

Answer Back

Test Final Test 4

ST program execution sequence
*The initial value of "uTotalProduction" is "100". The value of the variable "uTotalProduction" will be "101" after the following example program is processed. Select the correct "uTotalProduction" status after a few seconds elapse.

```
uTotalProduction := uTotalProduction + 1;
```

- ○ The value remains at 101.
- ◉ The value keeps changing.

Answer   Back

Test Final Test 5

Combining multiple conditions
Select the correct ST program example that represents the same operation as the following program example in LD.

```
bResult0 := (bResult0  OR bFlag1) AND bFlag2;
bResult1 := bResult0 AND bFlag3;
```

```
bResult0 := (bFlag0 OR bFlag2) AND bFlag1;
bResult1 := bResult0 AND bFlag3;
```

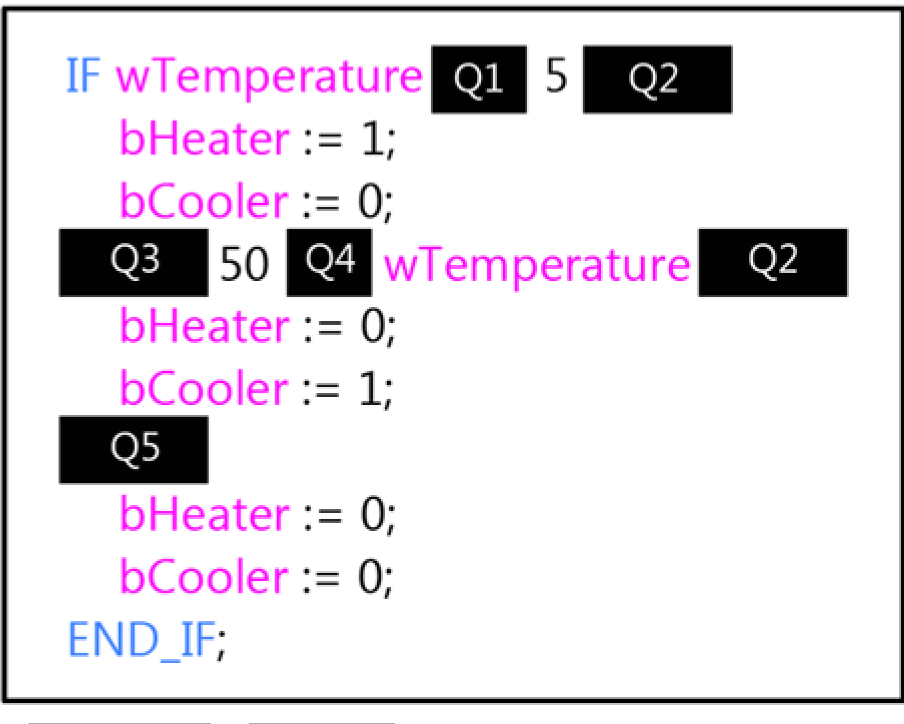
Answer Back

Test Final Test 6

Description of IF statements in ST
The following operation is executed by the example program below.

- If the temperature drops to 5 degrees or less, the heater turns on and the cooler turns off.
- If the temperature exceeds 50 degrees, the heater turns off and the cooler turns on.
- If the temperature does not apply to the above statements, both the heater and cooler are turned off.

*Variable names: Temperature (wTemperature), heater (bHeater), and cooler (bCooler)
Select the correct choice for each blank section of the example program.



Q1	<=
Q2	THEN
Q3	ELSIF
Q4	<
Q5	ELSE

Answer Back

Test Final Test 7

CASE statements
Select the correct one for each (Q1 to Q5) of the following description of CASE statements.

CASE statements are used for branching according to the value of (Q1).
In the following example program, when the value of (Q2) is 25, the variable (Q3) is assigned a value of (Q4).
When the value of variable (Q2) is not equal to 10, 25, or 8, the variable (Q3) is assigned a value of (Q5).

```
CASE wCode OF
    10:    uLane := 1;
    25:    uLane := 2;
    8:     uLane := 3;
   ELSE    uLane := 4;
END_CASE;
```

Q1 Integers

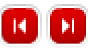
Q2 wCode

Q3 uLane

Q4 2

Q5 4

Answer Back



Test Final Test 8

ST arrays and repetitive statements

The following example program totals the planned production volume of all models destined for Country Y and then assigns this value to a variable. Select the section of the array that is read after the FOR statement is executed in a loop 3 times.

```
uProductionToday := 0;
FOR wCarModel := 0 TO 3 BY 1 DO
  uProductionToday := uProductionToday + uProduction[1,wCarModel];
END_FOR;
```

Array used to store the estimated number of units produced per model and destination (uProduction)

Model (column)

Destination (row)	Model 1	Model 2	Model 3	Model 4
Country X	[0,0]	[0,1]	[0,2] C	[0,3]
Country Y	[1.0]	[1.1] A	[1.2] D	[1.3] E
Country Z	[2,0]	[2,1] B	[2,2]	[2,3]

- ○ A
- ○ B
- ○ C
- ● D

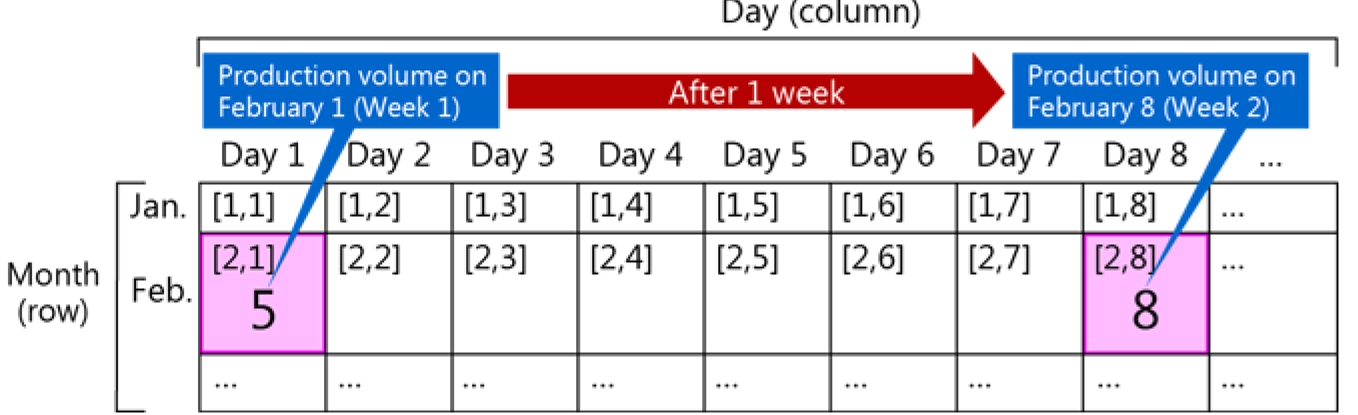
Answer Back

Test | Final Test 9

ST arrays and repetitive statements
The following example program obtains the total production volume on the same days of the week. The total over 4 weeks is obtained from the array that stores the production volume per day. Select the correct figure for the example program.

```
uTotalProduction := 0;
FOR wOnceAWeek := 1 TO ■ BY 7 DO
  uTotalProduction := uTotalProduction + uProductionByDate[2,wOnceAWeek];
END_FOR;
(* Extracts and totals the production volume on the same days of the week over 4 weeks starting from February 1. *)
```

Array that stores the production volume per day (uProductionByDate)



- (●) 22
- ( ) 21
- ( ) 4
- ( ) 28

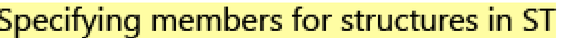
Answer | Back

Test Final Test 10

Characteristics of structures in ST
Select an incorrect description of structures.

- ○ Structures are used to organize and store data on devices by conditions such as status and specifications.
- ○ Programs that process large amounts of data can be concisely written using structures.
- ● All members defined in the structure must have the same data type.
- ○ Values can be assigned to the members in the same structure without being individually specified.

Answer   Back



Final Test 11

Specifying members for structures in ST
The following structure organizes variables related to an automobile production line.
Select the correct description for specifying the member "bStatus" in this structure.

Parameter	Variable name
Mode	sModel
Status	bStatus
Target production for the current day	uPlanQty
Current production number	uActualQty

Structure

- (•) st3rdLine.bStatus
- ( ) st3rdLine->bStatus
- ( ) st3rdLine[bStatus]
- ( ) st3rdLine[1]

Answer | Back

Test Final Test 12

Handling strings in ST

The following example program extracts a specific string from the string "e3211151602" stored in the variable "sBarcodeData". The MID function extracts the specified number of characters starting at the specified start position.

Select the string extracted correctly.

Number of characters to extract

Start position to extract a string

```
sData := MID(sBarcodeData,4,4) ;
(* Extracts the text string from "e3211151602". *)
```

- ○ 1151
- ○ 1602
- ○ e321
- ● 1115

Answer Back

Test | Test Score

You have completed the Final Test. You results area as follows.
To end the Final Test, proceed to the next page.

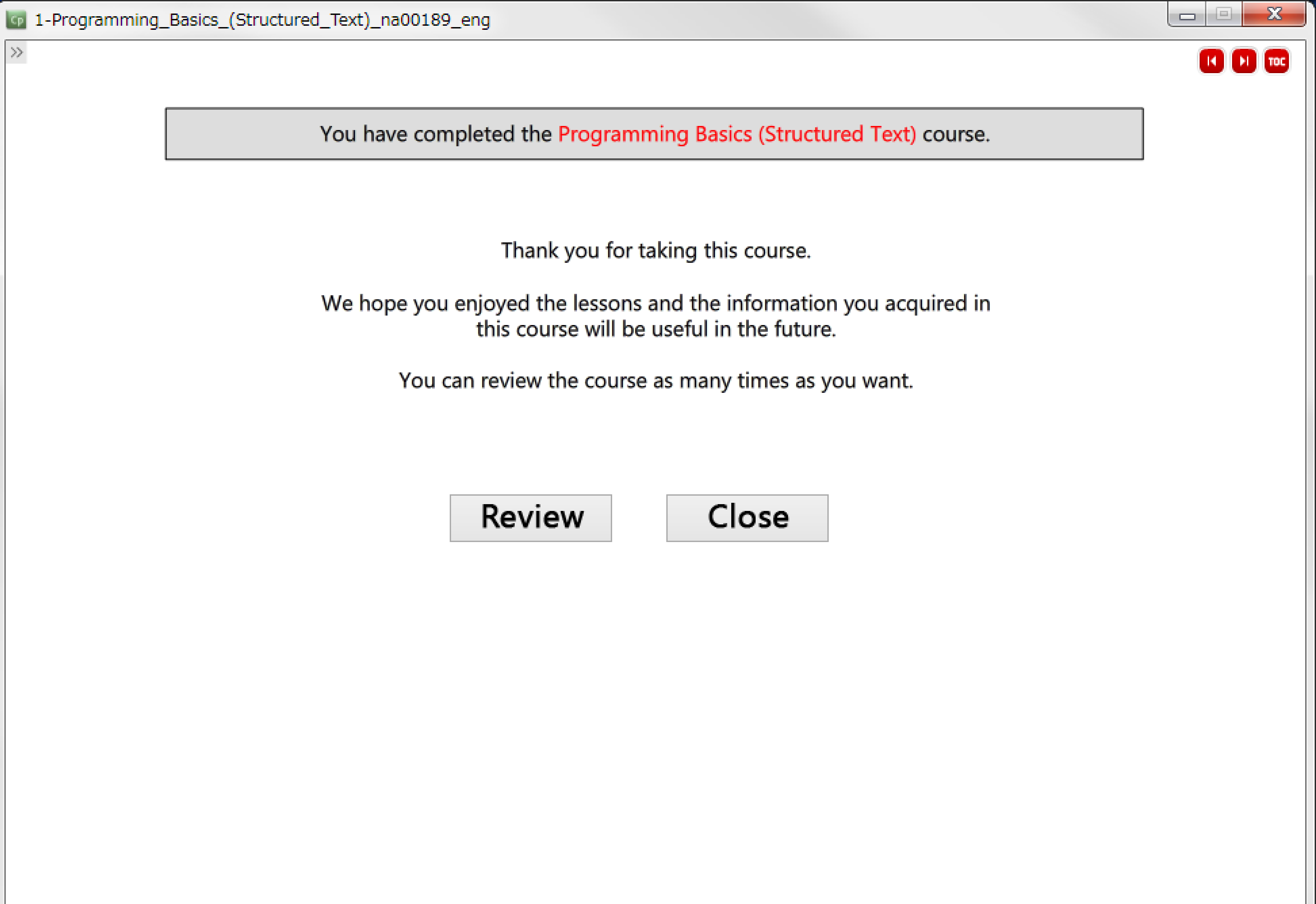
Correct answers : 12

Total questions : 12

Percentage : 100%

Proceed   Review

Congratulations. You passed the test.


1-Programming_Basics_(Structured_Text)_na00189_eng
You have completed the Programming Basics (Structured Text) course.
Thank you for taking this course.
We hope you enjoyed the lessons and the information you acquired in this course will be useful in the future.
You can review the course as many times as you want.
Review
Close