MODULE 5: ASSIGNMENT STATEMENTS AND EXPRESSIONS

Module Overview

Assignment statements are among the simplest statements in C/AL. There are several types of expressions, such as string expressions, numeric expressions, relational, and logical expressions. Assignment statements and expressions are among the most common building blocks of the C/AL code.

Understanding assignment statements and expressions help you write effective C/AL code.

Objectives

- Explain the concepts of assignment, statement, and assignment statement.
- Describe the syntax of statements and introduce the statement separator.
- Describe automatic type conversions for string, numeric, and other data types.
- Use assignment statements and the Symbol Menu.
- Explain the concepts of expressions, terms, and operators.
- Describe the syntax of an expression.
- Describe the string operator.
- Use the string operator.
- Describe the MAXSTRLEN and the COPYSTR functions.
- Use the MAXSTRLEN and the COPYSTR functions in an expression.
- Define numeric expressions, arithmetic operators, and operator precedence.
- Describe the arithmetic operators, and provide examples.
- Use the arithmetic operators and examine the operator precedence.
- Define relational and logical operators and expressions.
- Describe the use of relational expressions for comparison.
- Describe the use of relational expressions for set inclusion.
- Describe the use of logical expressions.
- Use logical and relational expressions in a page.

Assignment Statements

Assignment statements assign a value to a variable. This means that a variable can represent different values at different times.

Assignment

Assignment means to set a variable to a value. When a variable has a value, it keeps that value until another value is assigned to it, or until the current set of code ends and the system no longer keeps track of the variable. C/AL has several assignment methods. One of these is the simple assignment statement.

Statement

A statement in programming context is a single complete programming instruction that is written as code. Think of a statement as a code line, because you typically write one statement per line. However, one statement may span several actual lines of the C/AL code, and a single line of code sometimes may include several statements.

Assignment Statement

The assignment statement is a specific type of statement that specifically assigns a value to a variable.

The Syntax of Statements

The ability to assign different values to variables is a cornerstone of programming. Different programming languages have different syntax for assigning values to variables.

Function Call Statement

Certain function calls, such as the **MESSAGE** function, can be a statement on their own. They are known as *function call* statements.

The syntax of a function call statement is as follows:

Code Example

```
[return value :=] <Function Call>
```

The result of a function call varies depending on the function that is called. Similarly, the syntax of the function call itself varies with the function that is called.

Assignment Statement

The syntax of an assignment statement is almost as easy as the syntax of the function call statement. The syntax is as follows:

Code Example

```
<variable> := <expression>
```

The assignment statement evaluates the expression and assigns the resulting value to the variable. The terms *evaluation* and *expression* are described in upcoming lessons. For now, use either a constant or another variable on the right side of the colon equals, as a simple expression.

Assignment Operator

The "colon equals" (:=) is known as the assignment operator. You use it to assign a value or an expression that evaluates to a value to a variable.

The Statement Separator

A single programming statement may span several code lines; one code line may consist of multiple statements. Therefore, the C/AL compiler must be able to detect when one statement ends and another statement begins. The C/AL compiler recognizes a statement separator as the indicator of a new statement. The statement separator is the semicolon (;).

The following example shows the syntax of a trigger:

Code Example

```
[ <statement> { ; <statement> } ]
```

Use brackets to indicate that whatever the brackets enclose is optional. It can be present or not. Braces indicate optional parts. Whatever the braces enclose is optional and can be repeated zero (0) or more times. In other words, it is optional for any statements to appear in a trigger. As many statements as necessary can appear in the trigger, as long as each statement is separated from the other statements by a semicolon.

A statement must follow a semicolon. If the last statement in a trigger ends with a semicolon, the C/AL compiler automatically inserts a Null statement after the last semicolon and before the end of the trigger. A *null statement* is an expression statement with the expression missing. Use a null statement when the syntax calls for a statement but no expression evaluation. It consists of a semicolon. A semicolon does not signal the end of a statement (it is not a statement terminator), but instead it signals the beginning of a new statement (it is a statement separator). This is an important difference in understanding the syntax of other statements.

Automatic Type Conversions

Before you can assign a variable to a value, the type of the value must match the type of the variable. However, within limits, you can automatically convert certain types in an assignment operation.

String Data Types

You can convert variables of string data types (code and text) automatically from one to the other. For example, if **Description** is a variable of type Text, and **CodeNumber** is a variable of type Code, the following statement is valid:

Code Example

```
CodeNumber := Description
```

The text value in the **Description** variable is converted into code before it is assigned to the **CodeNumber** variable. This means that all lowercase letters are converted to uppercase, and all leading and trailing spaces are deleted. Therefore, the value that is assigned to the code variable is of type Code. This conversion process does not affect the value that is stored in the **Description** variable. Variables on the right side of the assignment operator are not changed by the assignment operation. Only the variable on the left side of the assignment operator is changed.

Note: *Automatic conversion does have limitations. For example, if the value of the* ***Description*** *text variable has more characters than the length of the* ***CodeNumber*** *variable, an error occurs when the program runs and executes this statement. This type of error is known as a* run-time error.

Numeric Data Types

Variables of the numeric data types (integer, decimal, option, and char) can convert automatically from one to another, with the following several restrictions as follows:

- A decimal value must be a whole number (without any value after the decimal separator) to convert to any of the other numeric data types.

- The value must fall within the range of the variable type. For example, you cannot automatically convert a decimal to an integer unless the value is between -2,147,483,648 and 2,147,483,647 which is the valid range for integers. Also, you cannot automatically convert an integer to a char, unless the value is between 0 and 255.

***Note:** The C/AL Compiler does not know if the conversion will succeed at run time. Therefore, the C/AL compiler always enables you to write code that implies automatic numeric data-type conversion. However, if either of these is violated while program is running, a run-time error occurs.*

Other Data Types

Variables of string data types and numeric data types can be automatically converted from one to the other. However, no other variable types can be automatically converted during assignment operation.

***Note:** If you try to write an assignment statement that assigns variables between incompatible types, such as assigning a Boolean value to a Text variable, a* compile-time error *occurs.*

Use Assignment Statements and the Symbol Menu

The following demonstrations show how to use the assignment statements on several data types and introduce the **C/AL Symbol Menu**.

Demonstration: Create a Simple Assignment Statement

The following demonstration shows how to use a simple assignment statement.

Demonstration Steps

1. Design codeunit **90000**, **My Codeunit** from the Object Designer.
   a. In **Object Designer**, click **Codeunit**.
   b. Select codeunit 90000, **My Codeunit**, and then click **Design**.

2. Delete all code in the **OnRun** trigger.
   a. In the **OnRun** trigger, select all code by positioning the cursor on the first row, and then pressing CTRL+SHIFT+END.
   b. Delete all code, by pressing Del, or F4.
   c. Click **Yes** to confirm the deletion.

3. In the **OnRun** trigger, enter the code that assigns value of 25 to the **LoopNo** variable, and then displays the variable name and value on screen.
   a. In the **OnRun** trigger, enter the following code:

```
LoopNo := 25;
MESSAGE('The value of %1 is %2','LoopNo',LoopNo);
```

4. Compile, save, close, and run the codeunit.
   a. Click **File** > **Save**.
   b. In the **Save** dialog window, verify that the **Compiled** check box is selected, and then click **OK**.
   c. Close the **C/AL Editor** window.
   d. In **Object Designer**, select codeunit 90000, **My Codeunit**.
   e. Click Run, and then see the results.

Demonstration: Create Multiple Messages

The following demonstration shows how you can assign different types of values to different types of variables, and how to specify constants of various data type in C/AL Editor.

Demonstration Steps

1. Design codeunit **90000**, **My Codeunit** from the Object Designer.
   a. In **Object Designer**, click **Codeunit**.
   b. Select codeunit 90000, **My Codeunit**, and then click **Design**.

2. In the **OnRun** trigger, add the code that assigns different values to different variables according to the data type. The code then displays the variable names and values.
   a. In the **OnRun** trigger, enter the following code:

```
LoopNo := -30;
MESSAGE('The value of %1 is %2','LoopNo',LoopNo);
Amount := 27.50;
MESSAGE('The value of %1 is %2','Amount',Amount);
"When Was It" := 093097D;
MESSAGE('The value of %1 is %2','When Was It',"When Was It");
"Code Number" := ' abc 123 x';
MESSAGE('The value of %1 is %2','Code Number',"Code Number");
```

3. Compile, save, close, and run the codeunit.
   a. Click **File** > **Save**.
   b. In the **Save** dialog window, verify that the **Compiled** check box is selected, and then click **OK**.
   c. Close the **C/AL Editor** window.
   d. In **Object Designer**, select codeunit 90000, **My Codeunit**.
   e. Click **Run**, and then see the results.

Note: *MESSAGE functions do not stop the execution. They store the messages added to a queue and display them after the processing has completed. Messages do not interrupt processing. This means that you cannot use messages to give users feedback on the processing that is in progress. You can use messages only to give users the processing results.*

Demonstration: Use the Symbol Menu

The **C/AL Symbol Menu** displays variables, functions, and objects that are defined in the **C/AL Globals** window. The C/AL menu also provides information about the syntax and description of the variables, functions, and objects. The following demonstration shows the **C/AL Symbol Menu**, and explains how to use it.

Demonstration Steps

1. Design codeunit **90000**, **My Codeunit** from the Object Designer.
   a. In **Object Designer**, click **Codeunit**.
   b. Select codeunit 90000, **My Codeunit**, and then click **Design**.
   c. Go to a new line at the end of the code in the **OnRun** trigger.

2. Open the **C/AL Symbol Menu**.
   a. Do any of the following to open the **C/AL Symbol Menu**:
      - On the **View** menu, click **C/AL Symbol Menu**
      - Press **F5**
      - Click the **C/AL Symbol Menu** button on the Toolbar.

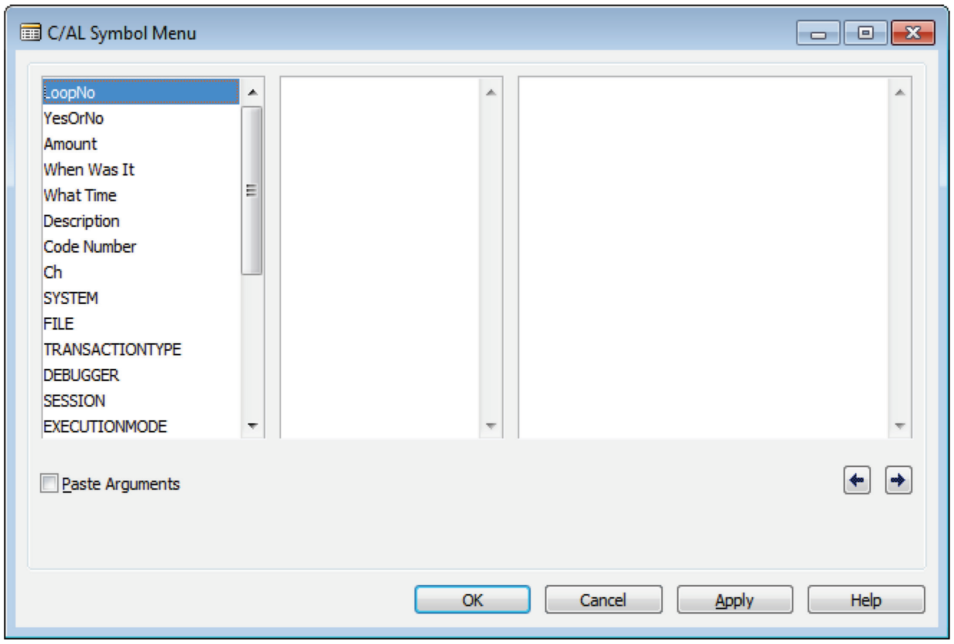


FIGURE 5.1: THE C/AL SYMBOL MENU

Note: *The left panel contains a list of all identifiers that are defined by the developer, and the list of categories of system-provided functions and objects.*

3. Use the **C/AL Symbol Menu** to assign value of **TRUE** to the **YesOrNo** variable.
   a. Select the line that says **YesOrNo**, and then click **OK**. The **C/AL Symbol Menu** closes and the word **YesOrNo** is displayed in the code.
   b. After YesOrNo, type the following ":= TRUE;"

   After you complete the line, it should read as follows:

```
YesOrNo := TRUE;
```

4. Use the **C/AL Symbol Menu** to write a line of code that displays the name and the value of the **YesOrNo** variable.
   a. On the first empty line, enter the following code:

```
MESSAGE('The value of %1 is %2','
```

   b. With the cursor immediately after the single quotation mark, press F5 again.
   c. Verify that **YesOrNo** is selected in the **C/AL Symbol Menu**, and then click **OK**.
   d. Type a single quotation mark (') and a comma (,).

e. Press F5 again, and then press ENTER. This is the same as clicking **OK** on the **C/AL Symbol Menu**.

f. Type a closing parenthesis ()) and a semicolon (;). The result is as follows:

```
YesOrNo := TRUE;

MESSAGE('The value of %1 is %2','YesOrNo',YesOrNo);
```

5. Use the **C/AL Symbol Menu** to write code that assigns a value to the **Description** field, and displays it on screen.
    a. Use the same method for **Description**, but set the value (located after the Assignment Operator) to 'Now is the time.' The result is as follows:

```
Description := 'Now is the time. ';

MESSAGE('The value of %1 is %2',

'Description',Description);
```

6. Use the **C/AL Symbol** menu to assign a value to the **"What Time"** time variable. It then displays it on screen.
    a. Again, use the **C/AL Symbol Menu** to enter another two lines that sett and display the **"What Time"** variable. Set its value to 153000T.

***Note:** By using the **Symbol Menu**, the double quotation marks are automatically inserted when it is necessary. In this case, the double quotation marks inside the single quotation marks are not needed. Therefore, go back and remove them.*

The result is as follows:

Code Example

```
"What Time" := 153000T;

MESSAGE('The value of %1 is %2','What Time',"What Time");
```

7. Compile, save, close, and run the codeunit.
    a. Click **File** > **Save**.
    b. In the **Save** dialog box, verify that the **Compiled** check box is selected, and then click **OK**.
    c. Close the **C/AL Editor** window.

   d. In **Object Designer**, select codeunit 90000, **My Codeunit**.
   e. Click Run, and then see the results.

Demonstration: Set Char Constants

Because of automatic type conversion, a Char variable can be set by using either a number or a one character text. The following demonstration shows how Char variables are assigned, and how they are displayed on screen.

Demonstration Steps

1. Design codeunit **90000**, **My Codeunit** from the Object Designer.
   a. In **Object Designer**, click **Codeunit**.
   b. Select codeunit 90000, **My Codeunit**, and then click **Design**.

2. In the **OnRun** trigger, write code that assigns the value to the **Ch** variable, by using numeric and character constants.
   a. In the **OnRun** trigger, append the following code:

Code Example

```
Ch := 65;
MESSAGE('The value of %1 is %2','Ch',Ch);
Ch := 'A';
MESSAGE('The value of %1 is %2','Ch',Ch);
```

3. Compile, save, close, and run the codeunit.
   a. Click **File** > **Save**.
   b. In the **Save** dialog box, verify that the **Compiled** check box is selected, and then click **OK**.
   c. Close the **C/AL Editor** window.
   d. In **Object Designer**, select codeunit 90000, **My Codeunit**.
   e. Click Run, and then see the results.

Note: *Both 65 and 'A' result in the same value shown in the message. This is because 65 is the ASCII code for the uppercase A. Experiment with other numbers to see results.*

Demonstration: Set Option Constants

Option variables map to integers. You can assign a value to an option variable by using a numeric value. In C/AL, you can also write option constants and assign option variables by using special syntax. The following demonstration shows how to assign values to option variables.

Demonstration Steps

1. Design codeunit **90000**, **My Codeunit** from the Object Designer.
   a. In **Object Designer**, click **Codeunit**.
   b. Select codeunit 90000, **My Codeunit**, and then click **Design**.

2. In the **OnRun** trigger, write code that assigns a value to the Color variable, from an integer constant, and from an option constant. The code should display the variable name and value on screen for each code type.
   a. In the **OnRun** trigger, append the following code:

```
Color := 2;
MESSAGE('The value of %1 is %2','Color',Color);
Color := Color::Yellow;
MESSAGE('The value of %1 is %2','Color',Color);
```

3. Compile, save, close, and run the codeunit.
   a. Click **File** > **Save**.
   b. In the **Save** dialog window, verify that the **Compiled** check box is selected, and then click **OK**.
   c. Close the **C/AL Editor** window.
   d. In **Object Designer**, select codeunit 90000, **My Codeunit**.
   e. Click Run, and then see the results.

Demonstration: Compile-Time and Run-Time Errors

Not every value can be converted automatically to a value of another type. Errors result if you try to convert types that cannot be converted automatically. Some obvious errors, such as assigning a text constant to an integer variable, are detected and reported by the C/AL compiler, whereas some less obvious errors, such as assigning a value from another variable outside the data type boundaries, are only detected and reported during run time.

The following demonstration shows the concepts of compile-time and run-time errors.

Demonstration Steps

1. Design codeunit **90000**, **My Codeunit** from the Object Designer.
   a. In **Object Designer**, click **Codeunit**.
   b. Select codeunit 90000, **My Codeunit**, and then click **Design**.

2. In the OnRun trigger, append the code which assigns both correct and incorrect values. Compile, save, and run the codeunit after every line that you write.
   a. In the **OnRun** trigger, append the following code:

Add the lines one at a time. Compile, save, close, and run the codeunit between each line of code to view any error messages. After a compile time error occurs, delete that line of code.

```
Description := 'Now is the time. Here is the place.';
LoopNo := 27.5;
YesOrNo := 1;
Amount := 27.5;

LoopNo := Amount;
```

Note: *Only the first and the fourth lines actually compile, as the compiler does not compile code that it recognizes does not work.*

Because the first line contains a constant and variable of type text, the complier does not discover the overflow until the codeunit executes. The over happens because the Description is defined as 30 characters long, however, the constant is 36.

For the fourth line of code, the Amount contains a value that cannot be converted to an integer. This error was discovered when the codeunit executes.

These kinds of errors are known as run-time errors*. When a run-time error occurs, it stops all additional processing and produces an error message.*

Expressions, Terms, and Operators

Expressions, terms, and operators are part of statements. An operator operates on one or more terms that makes up to an expression which then evaluates to a value.

Expression

An expression specifies the information to generate a desired value. Similar to a variable and a constant, an expression has a type and a value. A constant's value is always known; a variable's value is determined at run time. The system must access memory for the value of constants and variables. However, an expression is evaluated at run time to determine its value. The following are examples of expressions:

- FunctionX + 7
- Quantity * UnitCost

Evaluation

To evaluate an expression means to follow the instructions set out in the formula exactly, and then determine the expression's type and value at that time. Depending on the values of the variables included in the expression, the value may be different each time that the expression is evaluated, although its type does not change.

Term

A term is the part of an expression that evaluates to a value. It can be a variable, a constant, or a function call, as long as the function returns a value. A term can also be another expression that is enclosed by parentheses. This kind of term is also known as a sub-expression. The previous examples of expressions have the following terms:

- FunctionX
- 7
- Quantity
- UnitCost

Operator

An operator is the part of an expression that acts upon either the term directly following it (for example, a unary operator), or the terms on either side of it (for example, a binary operator). Operators are represented by symbols, such as +, >, /, and =) or reserved words, such as **DIV** and **MOD**. They are defined by the C/AL and a developer cannot add or change them.

When an expression evaluates and the operator operates on its terms, it results in a value that may be the value of the expression, or a term that is used by another operator. Some examples of operators and their uses are as follows:

Example	Remarks
5 + (-8)	The + is a binary arithmetic operator, the - is a unary arithmetic operator.
TotalCost/Quantity	The / is a binary operator.
'cat' + ' and dog'	The + is now used as a binary string operator.
(Quantity > 5) OR (Cost <= 200)	The OR is a binary logical operator. The > and <= are binary relational operators.

The Expression Syntax

Because of its complexity, an expression's syntax requires several more syntax statements than you have seen previously. The following list shows valid syntax for several kinds of operators.

Operator Type	Valid Syntax
<unary operator>	+ | - | NOT
<string operator>	+
<arithmetic operator>	+ | - | * | / | DIV | MOD
<relational operator>	< | > | = | <= | >= | <> | IN
<logical operator>	NOT | OR | AND | XOR
<operator>	<string operator> | <arithmetic operator> | <relational operator> | <logical operator>
<simple term>	<constant> | <variable> | <function call> | (<expression>)
<term>	<simple term> | <unary operator> <simple term>
<expression>	<term> { <operator><term> }

The braces that enclose an element mean zero (0) or more repetitions of that element.

The String Operator

The plus sign (+) is the only string operator. This indicates concatenation. *Concatenation* is the operation that joins two or more strings together to make one string. The *concatenation operator* is a binary operator. This means that it operates on the term preceding it, and the term following it. Both terms must be strings either of type Code or Text. If both terms are of type Code, the resulting concatenation is of type Code. Otherwise,, it is of type Text.

Lesson Objectives

Describe the string operator.

Evaluating Expressions

The following code lines show an example of an expression and how it is evaluated.

Code Example

```
CodeA := 'HELLO THERE';
TextA := 'How Are You? ';
CodeB := CodeA + '. ' + TextA;
```

The third line of code is evaluated as follows:

- The value of **CodeA** is obtained. The constant value then is concatenated to the end. Because **CodeA** is a Code while the constant (a dot and a trailing space) is a Text, the result is a Text and the value is 'HELLO THERE. '. This value becomes the first term for the next concatenation operator.
- The value of **TextA** is obtained. This value then is concatenated to the end of the previously generated text. Because both values are Text, the result is also of type Text, and the value is 'HELLO THERE. How Are You?'. This is the end of the expression. Therefore, this result becomes the result of the expression, both the type and the value.
- When the expression is evaluated, the new value is assigned to the **CodeB** variable by using the assignment operator. Because the expression is of type Text and **CodeB** is of type Code, the result of the expression must be converted to Code by using the automatic type conversion. The result of this conversion is 'HELLO THERE. HOW ARE YOU?' which is assigned to **CodeB**.

Demonstration: Use the String Operator

The following demonstration shows how to use the string operator.

Demonstration Steps

1. Create a new codeunit, and save it as codeunit **90001**, **My Codeunit 2**.
   a. In **Object Designer**, click **Codeunit**.
   b. Click **New**. This opens the **C/AL Editor** window.
   c. Click **File** > **Save**.
   d. In the **Save As** dialog window, in the **ID** field, enter “90001”, in the **Name** field, enter “My Codeunit 2”.
   e. Verify that the **Compiled** check box is selected, and then click **OK**.

2. Define the following global variables: **CodeA** of type Code[30], **CodeB** of type Code[50], and **TextA** of type Text[50].
   a. Click **View** > **C/AL Globals**.
   b. On the **Variables** tab, enter the following information:

Name	DataType	Length
CodeA	Code	30
CodeB	Code	50
TextA	Text	50

   c. Close the **C/AL Globals** window.
3. In the **OnRun** trigger, enter code that does the following:
   - Assigns the “HELLO THERE” to **CodeA**, “How Are You” to **TextA**.
   - Assigns the result of concatenation of **CodeA**, “! “, and **TextA** to the **CodeB** variable.
   - Shows the value of the **CodeB** variable on screen.

   a. Type the following code in the **OnRun** trigger:

```
CodeA := 'HELLO THERE';
TextA := 'How Are You? ';
CodeB := CodeA + '! ' + TextA;
MESSAGE('The value of %1 is %2','CodeB',CodeB);
```

4. Compile, save, close, and run the codeunit.
   a. Click **File** > **Save**.
   b. In the **Save** dialog window, verify that the **Compiled** check box is selected, and then click **OK**.
   c. Close the **C/AL Editor**.
   d. In **Object Designer**, select the codeunit 90001, **My Codeunit 2**, and then click **Run**.

Function Calls in Expressions

When characters exceed the maximum length of a string variable, run-time errors occur. To prevent run-time errors, carefully design the program so that this error never occurs. However, it is not always possible to design the program to avoid errors that are caused by string length.

The MAXSTRLEN Function

The **MAXSTRLEN** is a function that returns the maximum defined length of a string variable.

This function has one parameter that is the string variable in question. The return value is of type Integer.

The COPYSTR Function

The **COPYSTR** is a function that copies a substring of any length, from a specific position in a string (text or code) to a new string.

This function has the following three parameters:

- The first parameter is the string to copy (original string).
- The second is the position of the first character to copy. For example, if the second parameter is 1, the function starts to copy from the first character. If the second parameter is 5, the function starts to copy from the fifth character.
- The third parameter is the number of characters to copy. This is optional. If it is not specified, the function copies from the start position that is specified in the second parameter, to the last character of the original string. If the number of characters to copy is more than the characters in the original string, the maximum number of characters of the original string is copied instead.

The return value is a string either of type Text or Code, depending on the type of the original string in the first parameter.

Demonstration: Use the MAXSTRLEN and the COPYSTR Functions

The following demonstration shows how to use the **MAXSTRLEN** and **COPYSTR** functions in an expression.

Demonstration Steps

1. Design codeunit 90001, **My Codeunit 2** from the Object Designer.
   a. In **Object Designer**, select the codeunit 90001, **My Codeunit 2**.
   b. Click **Design** to open the codeunit in the **C/AL Editor**.

2. Define the following global variables: **MaxChar** of type Integer, and **Description** of type Text.
   a. Click **View** > **C/AL Globals**.
   b. On the **Variables** tab, enter the following information, under the

already defined variables:

Name	DataType	Length
MaxChar	Integer	
Description	Text	30

c. Close the **C/AL Globals** window.

3. In the **OnRun** trigger, append the code that sets the **MaxChar** variable to the maximum length of the **Description** variable, and then shows the value of the **MaxChar** variable on the screen.
   a. In the **OnRun** trigger, enter the following code under the last line of code:

```
MaxChar := MAXSTRLEN(Description);

MESSAGE('The value of %1 is %2','MaxChar',MaxChar);
```

***Note:** The result is that **MaxChar** is set to 30 which is the length of the **Description** variable. By using the **MAXSTRLEN** function, you know how many characters a variable can hold . Therefore, you can prevent overflowing the variable by carefully writing the code.*

4. Under the last line of code, enter the code that does the following:
   - Assigns the result of the concatenation of the "The message is" and **CodeB** to the **Description** variable.
   - Displays the value of the **Description** variable on screen.

   a. Under the last line of code, enter the following code:

```
Description := 'The message is: ' + CodeB;

MESSAGE('The value of %1 is %2','Description',Description);
```

5. Compile, save, close, and run the codeunit.
   a. Click **File** > **Save**.
   b. In the **Save** dialog window, verify that the **Compiled** check box is selected, and then click **OK**.
   c. Close the **C/AL Editor**.
   d. In **Object Designer**, select the codeunit 90001, **My Codeunit 2**, and then click **Run**.

***Note:** The expected run-time error occurs. The assignment statement assigns the value of the expression on its right to the variable on its left. In this case, the expression results in a string that is longer than the maximum that the **Description** variable can hold (30 characters). To avoid this issue, use the COPYSTR function to assign only as many characters as the target variable can hold.*

6. Design codeunit 90001, **My Codeunit 2** from the Object Designer.
   a. In **Object Designer**, select the codeunit 90001, **My Codeunit 2**.
   b. Click **Design** to open the codeunit in the **C/AL Editor**.

7. Change the line that assigns the **Description** variable so that it only assigns as many characters as **Description** can hold. Use the **COPYSTR** and **MAXSTRLEN** functions.
   a. Go to the line that assigns the **Description** variable.
   b. Change the line as follows:

```
Description := COPYSTR('The message is: ' +

CodeB,1,MAXSTRLEN(Description));
```

8. Compile, save, close, and run the codeunit.
   a. Click **File** > **Save**.
   b. In the **Save** dialog window, verify that the **Compiled** check box is selected, and then click **OK**.
   c. Close the **C/AL Editor**.
   d. In **Object Designer**, select the codeunit 90001, **My Codeunit 2**, and then click **Run**.

***Note:** The run-time error no longer is displayed. Instead, the value of the **Description** variable only includes the characters that meet the 30 character maximum length limitation.*

How This Example Works

The expression is to the right of the assignment statement. Here is the original expression:

Code Example

```
COPYSTR('The message is: ' +

CodeB,1,MAXSTRLEN(Description))
```

To evaluate a function like **COPYSTR**, you must first evaluate each of its parameters. The first step is to obtain the value of **CodeB** and concatenate it (note the plus sign [+]) with the string constant. The result is as follows:

Code Example

```
COPYSTR('The message is: HELLO THERE. HOW ARE
YOU?',1,MAXSTRLEN(Description))
```

The next step is to evaluate the second parameter. Because this is a constant, it is straightforward.

After you evaluate the second parameter, evaluate the third parameter. This parameter is a function that returns the defined length of its parameter. This must be a variable, not an expression. The result is as follows:

Code Example

```
COPYSTR('The message is: HELLO THERE. HOW ARE YOU?',1,30)
```

The final step is to evaluate the **COPYSTR** function. In this case, **COPYSTR** function copies characters from the text in the first parameter, starting with the first character, the second parameter, and copying up to 30 characters, the third parameter. The result is as follows:

Code Example

```
'The message is: HELLO THERE. H'
```

Now that the expression is evaluated, the assignment can be performed:

Code Example

```
Description := 'The message is: HELLO THERE. H';
```

The **Description** variable has a new value.

Numeric Expressions

Numeric expressions are expressions that result in a numeric value. The individual terms in a numeric expression may not necessarily be numeric. Numeric expressions use at least one arithmetic operator. When a numeric expression is evaluated, the result is a numeric data type. This can be one of the following: decimal, integer, option, or char.

Arithmetic Operator

An *arithmetic operator* is used in numeric expressions to operate on numeric or non-numeric terms. Examples of arithmetic operators include addition, subtraction, multiplication, and division symbols.

Numeric Expression

A numeric expression is an expression that results in a numeric value. The following are examples of numeric expressions:

- 5 + 2 * 3
- 10 / 2

Operator Precedence

Operator precedence is the order that operators are evaluated in an expression. Operators with a higher precedence are evaluated before operators with a lower precedence. For example, the multiplication operator (*) has a higher precedence than the addition operator (+). Therefore, the expression 5 + 2 * 3 evaluates to 11, instead of 21. This is the same under the usual left to right rule.

Arithmetic Operators

There are six arithmetic operators in C/AL:

- Plus Operator (+)
- Minus Operator (-)
- Times Operator (*)
- Divide Operator (/)
- Integer Divide Operator (DIV)
- Modulus Operator (MOD)

The Plus (+) Operator

The *plus operator* is used for several purposes. It can be used as a unary or a binary operator.

If the terms on either side of the plus operator are both strings, the plus operator is used as a string operator, to concatenate the two strings. The plus operator that is used with the string term is not an example of an arithmetic operator, because the result is not numeric.

If the plus operator has no term in front of it, it is used as a unary operator. When it is used as a unary operator, its purpose is to leave the sign of the term following it unchanged. When it is used as a unary operator, it does nothing and is rarely used. If it is used, its purpose is to explicitly show that the value of the term is positive. The following is an example of an expression. The first example was created by using the plus operator as a unary operator. The second example shows the same expression without the plus operator which evaluates to the same result.

- IntVariable * +11
- IntVariable * 11

The plus operator is usually used as a binary operator. The purpose is to add the term following it to the term preceding it. Both terms can be numeric, or one term can be a numeric and the other can be a date or a time. The expression results to a different data type, depending on the type of the terms.

The following rules apply to the plus operator:

- If either term is a decimal value, the result is decimal.
- If both terms are char values and the sum is less than 256, the result is char; otherwise, the result is integer.
- If both terms are option or integer values and the sum is in the allowed values for integers, the result is integer; otherwise, the result is decimal.
- If the term preceding the operator is a date and the term following the operator is an integer, the result is a date that is the integer number of days away from the date term. Therefore, the result of 03202001D + 7 is 03272001D. If the resulting value is an invalid date, a run-time error occurs.
- If the term preceding the operator is a time and the term following the operator is an integer, the result is a time that is the integer number of milliseconds away from the time term. Therefore, the result of 115815T + 350000 is 120405T. If the resulting value is an invalid time, a run-time error occurs.

The plus operator that is used with a date or a time term is not an example of an arithmetic operator, because the result is not numeric. The following list shows combinations of the preceding and following terms, and their results.

Plus (+)	Char	Option	Integer	Decimal	Date	Time
Char	Char	Integer	Integer	Decimal	N/A	N/A
Option	Integer	Integer	Integer	Decimal	N/A	N/A
Integer	Integer	Integer	Integer	Decimal	N/A	N/A
Decimal	Decimal	Decimal	Decimal	Decimal	N/A	N/A
Date	Date	Date	Date	N/A	N/A	N/A

Plus (+)	Char	Option	Integer	Decimal	Date	Time
Time	Time	Time	Time	N/A	N/A	N/A

The left column indicates the type of the term that precedes the plus operator. The top row indicates the type of the term that follows the plus operator.

If the result is a char, but the value is not a valid char value, the result type changes to an integer. If the result is an integer but the value is not a valid integer value, the result type changes to a decimal.

The Minus (-) Operator

Similar to the plus operator, the *minus operator* can be used as a binary operator or a unary operator. When it is used as a unary operator, its purpose is to change the sign of the term following it.

When it is used as a binary operator, the minus operator's purpose is to subtract the term following it from the term preceding it. Both terms can be numeric, and both can be a date or a time, or when the following term is an integer, the preceding term can be a date or a time. The expression results to a different data type, depending on the type of the terms.

The following rules apply to the minus operator.

- If the first term is a date and the second is an integer, the result is a date that is the integer number of days before the date term. Therefore, the result of 02252001D - 7 is 02182001D. If the resulting value is an invalid date, a run-time error occurs.
- If the first term is a time and the second term is an integer, the result is a time that is the integer number of milliseconds before the time term. Therefore, the result of 115815T - 350000 is 115225T. If the resulting value is an invalid time, a run-time error occurs.
- If one date is subtracted from another, the result is the integer number of days between the two dates. If one time is subtracted from another, the result is the integer number of milliseconds between the two times.

The following list shows combinations of preceding and following terms, and their results.

Minus (-)	Char	Option	Integer	Decimal	Date	Time
Char	Char	Integer	Integer	Decimal	N/A	N/A
Option	Integer	Integer	Integer	Decimal	N/A	N/A
Integer	Integer	Integer	Integer	Decimal	N/A	N/A
Decimal	Decimal	Decimal	Decimal	Decimal	N/A	N/A

Minus (-)	Char	Option	Integer	Decimal	Date	Time
Date	Date	Date	Date	N/A	Integer	N/A
Time	Time	Time	Time	N/A	N/A	Integer

The left column indicates the type of the term preceding the minus operator. The top row indicates the type of the term following the minus operator.

If the result is a char, but the value is not a valid char value, the result type changes to an integer. If the result is an integer but the value is not a valid integer value, the result type changes to a decimal.

The Times (*) Operator

The *times operator* (or the *multiplication operator*) is used only as a binary operator. Its purpose is to multiply the numeric term preceding it by the numeric term following it. The following list shows combinations of the preceding and following terms, and their results.

Times (*)	Char	Option	Integer	Decimal
Char	Char	Integer	Integer	Decimal
Option	Integer	Integer	Integer	Decimal
Integer	Integer	Integer	Integer	Decimal
Decimal	Decimal	Decimal	Decimal	Decimal

The automatic conversion rules apply, from char to integer, and integer to decimal.

The Divide (/) Operator

The divide operator is used only as a binary operator. Its purpose is to divide the numeric term preceding it by the numeric term following it. The result of this division is always of type Decimal. If the second term is zero (0), a run-time error occurs.

The Integer Divide (DIV) Operator

The integer divide operator is used only as a binary operator. Its purpose is to divide the numeric term preceding it by the numeric term following it. The result type of this division is always of type Integer. If the second term is zero (0), a run-time error occurs. Any decimals that resulted from an ordinary division are dropped. Therefore, the result of 17 DIV 8 is 2, whereas the result of 17 DIV 9 is 1.

The Modulus (MOD) Operator

The modulus operator (or the remainder operator) is used only as a binary operator. Its purpose is to divide the numeric term preceding it by the numeric term following it by using the integer division method and then returning the remainder of that division. The result of this operation is always of type Integer. If the second term is zero (0), a run-time error occurs. The following shows examples of modulus operator usage.

- 17 MOD 8 = 1
- 17 MOD 9 = 8

The modulus operator requires two numbers. The first number is the one that is converted by using the modulus function. The second number represents the number system being used. By definition, the number system starts at zero and ends at the second number minus one. For example, if the second number is ten, the number system that is used is from zero to nine. Therefore, the modulus represents what the first number converts to, if the numbering system only had the number of values that are indicated by the second number, and the first number is forced to restart at zero.

The following example shows several modulus operations:

- 15 modulus 10 is 5 (because 9 is the last number available, 10 is represented by going back from the start, or zero, 11 is 1, 12 is 2, and so on)
- 6 modulus 10 is 6
- 10 modulus 10 is 0
- 127 modulus 10 is 7

The result is the same if the first number is divided by the second by using an integer only. The remainder is returned as the value.

Operator Precedence Levels

The three levels of operator precedence used for arithmetic operators are as follows:

- The highest level is the unary operator level. This includes both positive (+) and negative (-).
- The second is the multiplicative operator level. This includes multiplication (*), both kinds of divides (/, DIV), and modulus (MOD).
- The lowest precedence level is the additive operator level. This includes both addition (+) and subtraction (-) binary operators.

General evaluations of expressions go from left to right. However, if one operator has a higher precedence level than another, it is evaluated first. To override these, developers can create subexpressions by enclosing parts of an expression with parentheses. Sub-expressions are always evaluated first.

Demonstration: Use the Arithmetic Operators

The following demonstration shows how to use the arithmetic operators and examine the operator precedence.

Demonstration Steps

1. Create a new codeunit, and save it as codeunit **90002**, **My Codeunit 3**.
   a. In **Object Designer**, click **Codeunit**.
   b. Click **New**. This opens the **C/AL Editor** window.
   c. Click **File** > **Save**.
   d. In the **Save As** dialog window, in the **ID** field, enter "90002", in the **Name** field, enter "My Codeunit 3".
   e. Verify that the **Compiled** check box is selected, and then click **OK**.

2. Define the global variables:
   a. Click **View** > **C/AL Globals**.
   b. On the **Variables** tab, enter the following information:

Name	DataType
Int1	Integer
Int2	Integer
IntResult	Integer
Amt1	Decimal
Amt2	Decimal
AmtResult	Decimal

3. In the **OnRun** trigger, enter the code as shown in the detailed steps.
   a. In the **OnRun** trigger, enter the following code:

```
Int1 := 25 DIV 3;
Int2 := 25 MOD 3;
IntResult := Int1 * 3 + Int2;
MESSAGE('The value of %1 is %2','IntResult',IntResult);
Amt1 := 25 / 3;
Amt2 := 0.00000000000000001;
AmtResult := (Amt1 - Int1) * 3 + Amt2;
MESSAGE('The value of %1 is %2','AmtResult',AmtResult);
```

4. Save, compile, close, and run the codeunit.
   a. Click **File** > **Save**.
   b. In the **Save** dialog window, verify that the **Compiled** check box is selected, and then click **OK**.
   c. Close the **C/AL Editor**.
   d. In **Object Designer**, select the codeunit 90002, **My Codeunit 3**, and then click **Run**.

Note: *The result shows that the* ***IntResult*** *is 25 and* ***AmtResult*** *is 1.*

5. Design codeunit **90002**, **My Codeunit 3** from the Object Designer.
   a. In **Object Designer**, select the codeunit 90002, **My Codeunit 3**.
   b. Click **Design** to open the codeunit in the **C/AL Editor**.

6. In the **OnRun** trigger, enter the code as shown in the detailed steps.
   a. In the **OnRun** trigger, under the last line of code, enter the following code:

```
Int1 := 5 + 3 * 6 - 2 DIV -2;
MESSAGE('The value of %1 is %2','Int1',Int1);
```

7. Save, compile, close, and run the codeunit.
   a. Click **File** > **Save**.
   b. In the **Save** dialog window, verify that the **Compiled** check box is selected, and then click **OK**.
   c. Close the **C/AL Editor**.
   d. In **Object Designer**, select the codeunit 90002, **My Codeunit 3**, and then click **Run**.

Note: *The result is 24.*

Examine the Precedence Rules

These steps describe the precedence rules that are used to evaluate this code:

1. The times operator (*) is evaluated, multiplying its preceding term (3) by its following term (6). This results in a new term of 18, leaving the following:

Code Example

```
Int1 := 5 + 18 – 2 DIV -2
```

2. The integer divide operator (DIV) is evaluated, dividing its preceding term (2) by its following term (-2). This results in a new term of minus one.

Code Example

```
Int1 := 5 + 18 – (-1)
```

3. The plus operator (+) is evaluated, adding its following term (18) to its preceding term (5). This results in a new term of 23.

Code Example

```
Int1 := 23 – (-1)
```

The binary minus operator (-) is evaluated, subtracting its following term (-1) from its preceding term (23). This results in the value of the complete expression; 24 (23 minus a negative 1 is 24).

Demonstration: Add Sub-Expressions

The following demonstration shows how the sub-expressions alter the precedence during expression evaluation.

Demonstration Steps

1. Design codeunit **90002**, **My Codeunit 3** from the Object Designer.
   a. In **Object Designer**, select the codeunit 90002, **My Codeunit 3**.
   b. Click **Design** to open the codeunit in the **C/AL Editor**.

2. In the **OnRun** trigger, enter the code as shown in the detailed steps.
   a. In the **OnRun** trigger, change the assignment for the **Int1** variable, as follows:

```
Int1 := (5 + 3) * (6 - 2) DIV -2;
```

3. Save, compile, close, and run the codeunit.
   a. Click **File** > **Save**.
   b. In the **Save** dialog window, verify that the **Compiled** check box is selected, and then click **OK**.
   c. Close the **C/AL Editor**.
   d. In **Object Designer**, select the codeunit 90002, **My Codeunit 3**, and then click **Run**.

Note: *The result is -16.*

Examine the Effect of Sub-Expressions

Because sub-expressions are evaluated first, this code evaluates as follows:

1. The first sub-expression is evaluated. The plus operator (+) adds its following term (3) to its preceding term (5), and results in the sub-expression value of 8. This value now becomes a term of the complete expression.

Code Example

```
Int1 := 8 * (6 – 2) DIV -2;
```

2. The next sub-expression is evaluated. The binary minus operator (-) subtracts its following term (2) from its preceding term (6) and results in the value of 4 for the sub-expression. This value is now a term of the complete expression.
3. The unary minus operator (-) and its following term (2) is evaluated and results in a value of negative 2 (-2).
4. The times operator (*) is evaluated, multiplying its preceding term (8) by its following term, (4) and results in a new term of 32.

Code Example

```
Int1 := 32 DIV -2;
```

The integer divide operator (DIV) is evaluated, dividing its preceding term (32) by its following term (-2). This results in the value of the complete expression: negative 16 (-16).

Code Example

```
Int1 := -16;
```

Relational and Logical Expressions

Logical and relational expressions result in a Boolean value. These expressions use logical or relational operators to determine their value. When logical and relational expressions are evaluated, the result is always either TRUE or FALSE.

Relational Operator

A *relational operator* is used in a relational expression to test a relationship between the term preceding it, and the term following it. The available relational operators are as follows:

- = (equal to)
- < (less than)
- > (greater than)
- <= (less than or equal to)
- >= (greater than or equal to)
- <> (not equal to)
- IN (included in set)

Relational Expression

A *relational expression* is an expression that compares values and results in a Boolean value. The individual terms in a relational expression usually are not of type Boolean. However, they must be comparable with one another. For example, an integer is comparable with a decimal, because both are numeric. But an integer is not comparable to a text, because one is numeric and the other is string. The following are examples of relational expressions:

- 5 <= IntVar
- DecVar <> IntVar

Logical Operator

A *logical operator* is used in a logical expression with one or two Boolean terms. The available logical operators are as follows:

- AND
- OR
- XOR
- NOT

Except for NOT, which is a unary operator, the rest of the logical operators are binary operators.

Logical Expressions

A *logical expression* is also an expression that compares values and results in a Boolean value. The difference between a relational expression is and a logical expression is that the terms in a logical expression must all be of type Boolean. The following are examples of logical expressions:

- TRUE AND FALSE (This logical expression has two terms and it results to FALSE.)
- FALSE OR NOT FALSE (This logical expression has two terms and two logical operators and it results to TRUE.)
- (Quantity > 5) OR (Quantity <= 10) OR (Price < 100) (This expression is a combination of relational and logical expressions.)

Relational Expressions for Comparison

Except for IN, all relational operators compare two values. These two values must be of the same type, or of comparable types. All the numeric types (for example, integer and decimal) are comparable. Both the string types (text and code) are also comparable.

Numeric Comparisons

If a comparison is performed between two numbers, the numeric rules apply. The following are examples of relational expressions for numeric comparisons:

- 57 = 57 is TRUE
- 57 = 58 is FALSE
- 57 < 58 is TRUE
- 57 <= 58 is TRUE
- 57 > 58 is FALSE
- 57 >= 57 is TRUE
- 57 <> 58 is TRUE

String Comparisons

In Microsoft Dynamics NAV Development Environment, string comparisons use a modified alphabetical order, not the ASCII order.

One difference is that special characters in languages such as the ø in Danish or the ñ in Spanish are positioned in their correct alphabetical order and not relegated to the end as they are in ASCII order.

Another difference is that the digits are positioned after the letters in alphabetical order, whereas in ASCII order, they are positioned before the letters.

Finally, in alphabetical order, the lowercase letters are positioned before the uppercase letters, whereas in ASCII order, the lowercase letters are positioned after the uppercase letters. The following are examples of relational expressions for string comparisons:

Relational Expression	In Native Database
'X' = 'X'	TRUE
'X' = 'x'	FALSE
'ark' > 'arc'	TRUE
'arC' > 'arc'	TRUE
'10' > '2'	FALSE
'00' <= 'OO'	FALSE
'é' > 'f'	FALSE
'abc' < 'ab'	FALSE
' a' <> 'a'	TRUE

There are special rules when a code variable is used in a string comparison. All trailing and leading spaces are removed and all letters are converted to uppercase. In addition, code values that consist of only digits are right-justified before comparison. Therefore, in Microsoft Dynamics NAV® Development Environment, the fifth expression in the previous example ('10' > '2') evaluates to TRUE if those values are assigned into variables of type Code before comparison.

Date and Time Comparisons

Date and Time values are compared by using the general calendar rule. Dates (or times) that are in the future are greater than dates (or times) that are in the past. A Closing Date which represents the last second of the last minute of the last hour of the day, is greater than the Normal Date for the same day, and less than the Normal Date for the next day.

Boolean Comparisons

Boolean values usually are not compared by using relational operators. However, when they are, TRUE is considered greater than FALSE.

Relational Operator Precedence

Relational operators have the lowest precedence of any operator. Relational comparison is performed after the expressions on either side of the relational operator are evaluated.

Therefore, the following example evaluates to TRUE:

- 5 * 7 < 6 * 6

The expression on the left side of the relational operator (5 * 7) is first evaluated to 35. Then the expression on the right side of the relational operator (6 * 6) is evaluated to 36. Then the value 35 is compared to 36.

Relational Expressions for Set Inclusion

The IN relational operator is used to determine inclusion. It operates on two terms, and determines whether the preceding term is in the following term. The following term must be a list of values, or a set, to compare. This list is part of the relational expression and is known as a *set constant*.

Set Constant

There are no variables of type set, but there are constants of type set. A set constant consists of an opening bracket ([) that is followed by a list of values that are separated by commas, and are followed by a closing bracket (]). For example, a set of all the even numbers from one to ten looks as follows:

Code Example

```
[2,4,6,8,10]
```

In addition to individual values, a member of a set can also be a range of values. A set of all the numbers from one to twenty not evenly divisible by ten looks as follows:

Code Example

```
[1..9,11..19]
```

In addition, a specific value or a value that is used as part of a range can be an expression. A set of list of numbers from 10 to 20, but excluding the variable n (as long as n is from 10 to 20), looks as follows:

Code Example

```
[10..n-1,n+1..20]
```

IN Operator

The IN operator's operation checks whether the value of the term that precedes it is included in the term that follows it (the set). The following are examples of relational expressions for set inclusion:

- 5 IN [2,4,6,8,10] is FALSE
- 5 IN [2,4..6,8,10] is TRUE
- 10 IN [1..9,11..19] is FALSE
- 'M' IN ['A'..'Z'] is TRUE

Logical Expressions

A logical expression evaluates Boolean terms and results in a Boolean value. In some cases, the Boolean terms are results of relational expressions.

Logical Operator Results

The following list reviews logical operator results:

- The NOT operator is a unary operator that logically negates the term following it. This changes TRUE to FALSE and FALSE to TRUE.
- The AND operator results in TRUE if both of the terms on either side of it are TRUE and otherwise results in FALSE.
- The OR operator results in FALSE if both of the terms on either side of it are FALSE and otherwise results in TRUE.
- The XOR operator results in TRUE if both of the terms on either side of it are not the same and otherwise results in FALSE.

The truth tables summarize these facts as follows:

NOT	True	False
Results in:	False	True

OR	True	False
True	True	True
False	True	False

AND	True	False
True	True	False
False	False	False

XOR	True	False
True	False	True
False	True	False

FIGURE 5.2: THE LOGICAL OPERATOR TRUTH TABLES

Logical Operator Precedence

The NOT operator has the same precedence as the other unary operators. This means that the NOT operator is evaluated before any other operator in the same expression.

The AND operator has the same precedence as the multiplicative operators, whereas the XOR and OR operators have the same precedence as the additive operators. The relational operators have the lowest precedence of all.

The following table summarizes the operator precedence of all the operators covered to this point:

Type of Operator	Operator	Comments
Sub-expression or Terms	( ) [ ] . ::	Sub-expression in parentheses is evaluated first.
Unary	+ - NOT	Highest precedence in an expression.
Multiplicative	* / DIV MOD AND	
Additive	+ - OR XOR	
Relational	< <= = >= > <> IN	Lowest precedence in an expression.
Range	..	Used in Set Constants.

When you use logical expressions, you should realize that the relational operators are lower in precedence than the logical operators. For example, to see whether a variable **N** is between 10 and 20 exclusively, the following expression cannot be used:

1. N >= 10 AND N <= 20

This is because the first operator to be evaluated is the AND operator, and the preceding (10) and following (N) terms are an integer, and not Boolean terms. This causes an error. Instead, use parentheses to force the relational operators to be evaluated first, as the following example shows:

- (N >= 10) AND (N <= 20)

With the parentheses, the two relational expressions are evaluated first. This results in two Boolean terms. Next the logical expression with the AND logical operator and the two Boolean terms are evaluated.

Lab 5.1: Use Logical and Relational Expressions in a Page

Scenario

Isaac is a developer at CRONUS International Ltd. He wants to test the newly acquired knowledge about logical operators. He wants to create a page, add several controls and an action to it, and write code that calculates a Boolean variable through a relational expression.

Exercise 1: Create a New Page

Exercise Scenario

Isaac creates a new page, defines the variables and the layout of the page, and then adds an action that calculates a relational expression.

Task 1: Create a New Page

High Level Steps

1. Create a new blank page.
2. Save the page as 90005, **My Test Page 1**.

Detailed Steps

1. Create a new blank page.
   a. In **Object Designer**, click **Page**.
   b. Click **New** to create a new page.
   c. Verify that the **Create blank page** option is selected, and then click **OK** to create a blank page.

2. Save the page as 90005, **My Test Page 1**.
   a. Click **File** > **Save**.
   b. In the **Save As** dialog window, in the **ID** field, enter “90005”.
   c. In the **Name** field, enter “My Test Page 1”.
   d. Verify that the **Compiled** check box is selected, and then click **OK**.

Task 2: Add Variables to the Page

High Level Steps

1. Define two global Integer variables, named **Value1** and **Value2**, and a global Boolean variable, named **Result**.

Detailed Steps

1. Define two global Integer variables, named **Value1** and **Value2**, and a global Boolean variable, named **Result**.
   a. On the **View** menu, click **C/AL Globals**.
   b. On the **Variables** tab, enter the following information:

Name	DataType
Value1	Integer
Value2	Integer
Result	Boolean

   c. Close the **C/AL Globals** window.

Task 3: Add Controls to the Page

High Level Steps

1. Under the **ContentArea**, add a group that is named **General**.
2. Under this group, add two groups that are named **Input** and **Output**.
3. Add field controls for **Value1** and **Value2** to the **Input** group, and for **Result** to the **Output** group.
4. Compile and save the page.

Detailed Steps

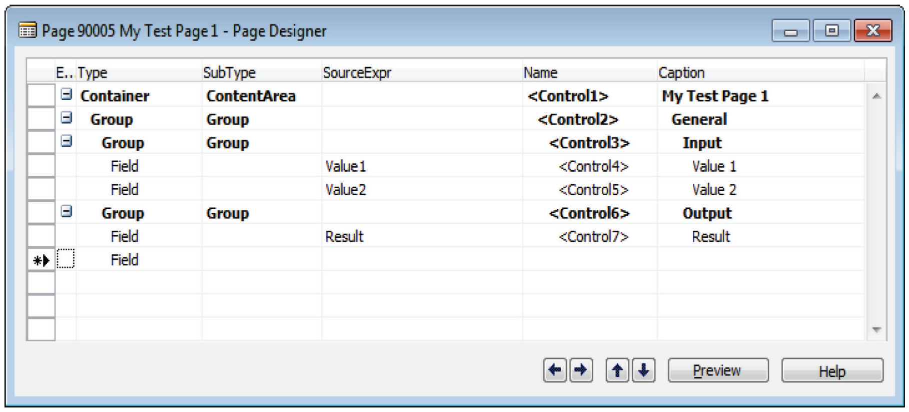
1. Under the **ContentArea**, add a group that is named **General**.
2. Under this group, add two groups that are named **Input** and **Output**.
3. Add field controls for **Value1** and **Value2** to the **Input** group, and for **Result** to the **Output** group.
   a. In the **Page Designer** window, enter the following information:

Type	SubType	SourceExpr	Caption
Container	ContentArea		My Test Page 1
Group	Group		General
Group	Group		Input
Field		Value1	Value 1
Field		Value2	Value 2
Group	Group		Output
Field		Result	Result

   b. Verify that the **General** group is indented one level under the **ContentArea** container.
   c. Verify that the **Input** and **Output** groups are indented one level under the **General** group.

d. Verify that the fields are indented one level under their parent groups.

The following figure shows the controls in the **My Test Page 1** page:



E..	Type	SubType	SourceExpr	Name	Caption
⊟	**Container**	**ContentArea**		**<Control1>**	**My Test Page 1**
⊟	**Group**	**Group**		**<Control2>**	**General**
⊟	**Group**	**Group**		**<Control3>**	**Input**
	Field		Value1	<Control4>	Value 1
	Field		Value2	<Control5>	Value 2
⊟	**Group**	**Group**		**<Control6>**	**Output**
	Field		Result	<Control7>	Result
	Field				

FIGURE 5.3: MY TEST PAGE 1 WINDOW

4. Compile and save the page.
   a. Click **File** > **Save**.
   b. In the **Save** dialog window, verify that the **Compiled** check box is selected.
   c. Click **OK**.

Task 4: Add an Action to the Page

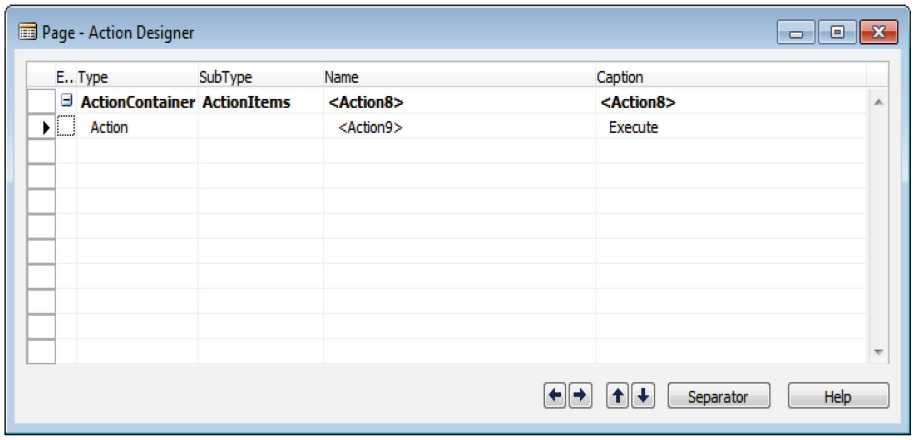
High Level Steps

1. In the ActionItems action container, create a new page action, and set its caption to **Execute**.
2. Add code that sets the value of **Result** to TRUE if **Value1** is higher than **Value2**, when a user clicks the **Execute** action.
3. Compile, save, and close the page.
4. Run the page to verify the results.

Detailed Steps

1. In the ActionItems action container, create a new page action, and set its caption to **Execute**.
   a. Click **View** > **Page Actions**.
   b. In the **Action Designer**, enter the following information:

Type	SubType	Caption
ActionContainer	ActionItems	<Action8>
Action		Execute



E..	Type	SubType	Name	Caption
⊟	**ActionContainer**	**ActionItems**	**<Action8>**	**<Action8>**
	Action		<Action9>	Execute

FIGURE 5.4: THE ACTION DESIGNER PAGE

2. Add code that sets the value of **Result** to TRUE if **Value1** is higher than **Value2**, when a user clicks the **Execute** action.
   a. Select the **Execute** action, and then press F9 to access the **C/AL Editor**.
   b. In the OnAction trigger, enter the following code:

Code Example

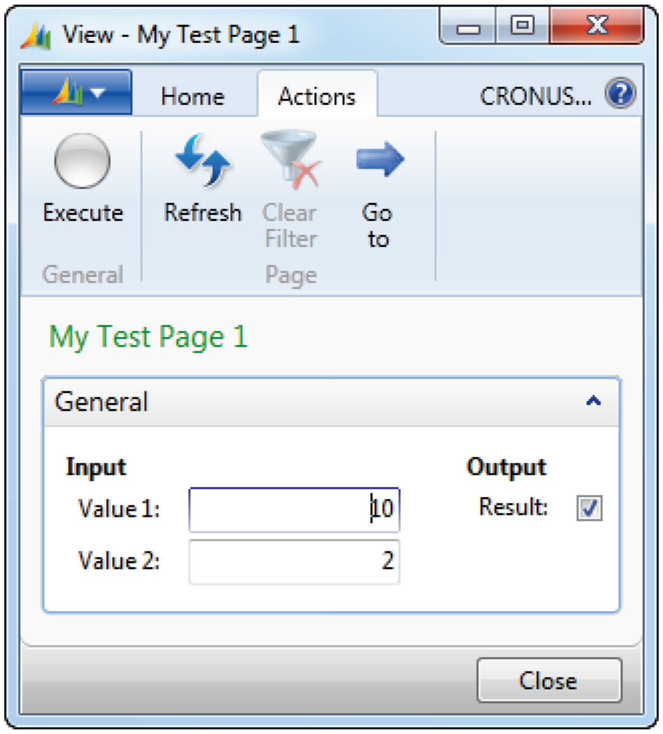
```
Result := Value1 > Value2;
```

   c. Close the **C/AL Editor**.
   d. Close the **Action Designer**.

3. Compile, save, and close the page.
   a. Click **File** > **Save**.
   b. In the **Save** dialog window, verify that the **Compiled** check box is selected.
   c. Click **OK**.
   d. Close the **Page Designer**.

4. Run the page to verify the results.
   a. In the **Object Designer**, select page 90005, **My Test Page 1**.
   b. Click **Run**.
   c. In the **My Test Page 1** page, in the **Value 1** field, enter "10", then in the **Value 2** field, enter "2".
   d. Verify that the **Result** check box is not selected.
   e. On the **Actions** tab of the ribbon, click **Execute**.
   f. Verify that the **Result** check box state changes to selected.

The following figure shows the **My Test Page 1** page after you click **Execute**:



Input

Value 1: 10

Value 2: 2

Output

Result: ✓

FIGURE 5.5: MY TEST PAGE 1 WINDOW

Module Review

Module Review and Takeaways

Assignment statements provide the basic foundation for developing an application by using C/AL. It lets developers assign values to variables. Assignment statements can have automatic type conversions.

There are many kinds of expressions in C/SIDE. These include string expressions, numeric expressions, relational, and logical expressions. Each type of expression is identified by its operator. For example, numeric expressions are identified by arithmetic operators; logical expressions are identified by logical operators. The operator precedence rule governs the use of all operators.

Assignment statements and expressions are pieces that consist of code in C/SIDE. Understanding assignment statements and different types of expressions help developers write effective code in C/SIDE.

Test Your Knowledge

Test your knowledge with the following questions.

1. What is the expression in this assignment statement:

   TextA := TextB;

2. In mathematics, the things that operators operate on are called *operands*. What are these things called in programming expressions?

3. What does the plus operator (+) do to text variables or constants?

4. What is the expression in this assignment statement:

   TextA := 'The ' + TextB;

5. What are the operators in this assignment statement:

   TextA := 'The ' + TextB;

6. What are the terms in this assignment statement:

   TextA := 'The ' + TextB;

7. Write down the result type and value of each of these expressions:
   - 57 * 10
   - 57 / 10
   - 57 MOD 10
   - 57 DIV 10
   - 9 / 4 – 9 DIV 4
   - (3 - 10) * - 5 - 10 + 2.5 * 4

8. For each of the following expressions, write down the value of the result of evaluating the logical or relational expression:

   - 5 * 7 > 3
   - 5 * -7 > -36
   - (3 > 5 - 1) OR (7 < 5 * 2)
   - (27 MOD 5 = 2) AND (27 DIV 5 = 5)
   - (5 > 3) XOR (7 = 7) AND (9 >= 10)
   - (10 > 2) AND ('10' > '2')
   - NOT (11 + 7 < 15) OR ('Great' > 'Greater') AND ('Less' < 'Lesser')
   - TRUE OR FALSE = TRUE

Test Your Knowledge Solutions

Module Review and Takeaways

1. What is the expression in this assignment statement:

   TextA := TextB;

   MODEL ANSWER:

   TextB

2. In mathematics, the things that operators operate on are called *operands*. What are these things called in programming expressions?

   MODEL ANSWER:

   Terms.

3. What does the plus operator (+) do to text variables or constants?

   MODEL ANSWER:

   The plus operator (+) concatenates text variables or constants.

4. What is the expression in this assignment statement:

   TextA := 'The ' + TextB;

   MODEL ANSWER:

   'The ' + TextB

5. What are the operators in this assignment statement:

   TextA := 'The ' + TextB;

   MODEL ANSWER:

   := and +

6. What are the terms in this assignment statement:

   TextA := 'The ' + TextB;

   MODEL ANSWER:

   'The ' and TextB.

7. Write down the result type and value of each of these expressions:

   - 57 * 10
   - 57 / 10
   - 57 MOD 10
   - 57 DIV 10
   - 9 / 4 – 9 DIV 4
   - (3 - 10) * - 5 - 10 + 2.5 * 4

   MODEL ANSWER:

Expression	Result	Result Data Type
57 * 10	570	Integer
57 / 10	5.7	Decimal
57 MOD 10	7	Integer
57 DIV 10	5	Integer
9 / 4 – 9 DIV 4	0.25	Decimal
(3 - 10) * - 5 - 10 + 2.5 * 4	35	Decimal

8. For each of the following expressions, write down the value of the result of evaluating the logical or relational expression:

   - 5 * 7 > 3
   - 5 * -7 > -36
   - (3 > 5 - 1) OR (7 < 5 * 2)
   - (27 MOD 5 = 2) AND (27 DIV 5 = 5)
   - (5 > 3) XOR (7 = 7) AND (9 >= 10)
   - (10 > 2) AND ('10' > '2')
   - NOT (11 + 7 < 15) OR ('Great' > 'Greater') AND ('Less' < 'Lesser')
   - TRUE OR FALSE = TRUE

   MODEL ANSWER:

Expression	Result
5 * 7 > 3	TRUE
5 * -7 > -36	TRUE
(3 > 5 - 1) OR (7 < 5 * 2)	TRUE
(27 MOD 5 = 2) AND (27 DIV 5 = 5)	TRUE
(5 > 3) XOR (7 = 7) AND (9 >= 10)	TRUE

Expression	Result
(10 > 2) AND ('10' > '2')	FALSE
NOT (11 + 7 < 15) OR ('Great' > 'Greater') AND ('Less' < 'Lesser')	TRUE
TRUE OR FALSE = TRUE	TRUE