

MODULE 6: C/AL STATEMENTS

Module Overview

There are many kinds of statements in C/AL. These statements can be grouped into conditional statements, repetitive statements, compound statements, and other statements that do not fall into these categorizations.

Conditional statements are statements that have one or more conditions. When you use these conditions, one or more other statements execute. There are several kinds of conditional statements in C/AL. To execute several other statements in conditional statements, these statements must be grouped by using a special construct. These statements are known as *compound statements*. *Repetitive statements* are statements that repeat several other statements multiple times. They are differentiated by the number of repetitions and how the number is determined. Repetitive statements can be used together with arrays. *Arrays* are complex variables that hold multiple values of the same data type. For example, use them to display addresses in a Microsoft Dynamics NAV 2013 report, or when you display information in a statistics page.

Understanding different kinds of statements help developers decide the best method to write code to achieve certain functionalities.

Objectives

- Define conditional statements and Boolean expressions.
- Describe the IF statement, the IF-THEN, and IF-THEN-ELSE syntax.
- Describe the EXIT statement and code indentation.
- Describe the CASE statement and the syntax of the CASE statement.
- Define compound statements and comments.
- Describe the syntax of compound statements with BEGIN and END.
- Understand the concepts of nested IF statements and the rule of correct indentation.
- Describe the syntax of comments.
- Use the IF, EXIT, CASE, and compound statements in a page.
- Test knowledge about C/AL statements.
- Define arrays and describe the components of arrays.
- Describe the syntax of arrays.
- Explain the power of arrays.
- Describe how to use strings as arrays of characters.

C/SIDE Introduction in Microsoft Dynamics® NAV 2013

- Introduce repetitive statements that are available in C/AL.
- Use arrays and repetitive statements in a page.
- Describe the WITH statement, record variables, and the syntax of the WITH statement.

Conditional Statements and Boolean Expressions

Understanding the concepts behind conditional statements and Boolean expressions helps lay the groundwork for understanding the **IF**, **CASE** and **EXIT** statements.

Conditional Statement

A conditional statement tests a condition and executes one or more statements based on the condition. The most frequently used conditional statement is the **IF** statement. Use it when there are only two possible values for the condition: *TRUE* or *FALSE*.

Boolean Expression

A Boolean expression results in a Boolean value, such as a Boolean variable or constant, a relational expression, or a logical expression.

The IF Statement

Use the **IF** statement to control if a part of the program executes. If the condition is *TRUE*, one set of one or more statements executes; and if the condition is *FALSE*, another set of one or more statements executes.

IF - THEN Syntax

Use the following syntax to execute a statement, if the condition is *TRUE*:

Code Example

```
IF <Boolean expression> THEN <statement>
```

A Boolean expression states a *TRUE* or *FALSE* condition. If the Boolean expression evaluates to *TRUE*, the statement following the reserved word, **THEN**, executes. If the Boolean expression evaluates to *FALSE*, the statement following **THEN**, does not execute.

For example, test whether an integer value is negative. If it is, change it to positive by using the following IF statement:

Code Example

```
IF Amount < 0 THEN  
  
    Amount := -Amount;
```

If the relational expression, *Amount < 0*, evaluates to *TRUE*, the assignment statement executes and the *Amount* becomes positive. If the relational expression evaluates to *FALSE*, the assignment statement does not execute and the *Amount* remains positive.



Note: The statement to execute if the relational expression evaluates to *TRUE*, is positioned under the **IF – THEN** line and indented by two spaces. This convention makes code easier to read. The Microsoft Dynamics NAV Development Environment works with or without this indentation. It works even if this statement appears on the first line, right after **THEN**.

IF - THEN - ELSE Syntax

Use the following syntax to execute a statement if the condition is *TRUE*, and to execute another statement if that condition is *FALSE*:

Code Example

```
IF <Boolean expression> THEN <statement 1> ELSE <statement 2>
```

A Boolean expression states a *TRUE* or *FALSE* condition. If the Boolean expression evaluates to *TRUE*, the statement following the reserved word, **THEN**, executes. If the Boolean expression evaluates to *FALSE*, the statement following the reserved word, **ELSE**, executes. Either of the two statements execute, but not both.

For example, to determine the unit price of an item, divide the total price by the quantity. However, if the quantity is zero, the division results in a run-time error. To prevent this, you must test whether the quantity is zero. If it is, instead of doing the division, give the unit price a valid value.

Code Example

```
IF Quantity <> 0 THEN  
  
    UnitPrice := TotalPrice / Quantity  
  
ELSE  
  
    UnitPrice := 0;
```

If the relational expression, *Quantity <> 0*, evaluates to *TRUE* (quantity is not zero [0]), the total price is divided by the quantity. The resulting value is assigned to the unit price. If the relational expression evaluates to *FALSE*, the unit price is set to zero.



Note: The two statements that optionally execute are indented by two spaces, and the **ELSE** reserved word is aligned with the **IF** reserved word. Even though the development environment works the same whether there are any spaces or new lines, this convention acts as a visual cue to which **ELSE** matches which **IF**. Also it assists in determining which condition triggers the **ELSE**.

The first assignment statement does not have a semicolon following it. This is because the **IF** statement is not finished yet, as the **ELSE** is a part of the **IF** statement. A semicolon (statement separator) suggests that a new statement is imminent, in this case, the **ELSE**, and there is no **ELSE**-only statement.

The EXIT Statement

Code in a trigger executes sequentially from the top to the bottom. Developers may not want to execute the rest of the code in a trigger. The **EXIT** statement is used for this purpose. When an **EXIT** statement executes, the program exits the current trigger and goes back to the object that calls the trigger, if any, or it goes back to the user.

Example of the Use of an EXIT Statement

The **EXIT** statement is frequently used with the **IF** statement to stop executing code under certain conditions. In the earlier code example, instead of testing whether the quantity is not zero, you could use the **EXIT** statement to skip the assignment statement, if the quantity is, in fact, zero.

Code Example

```
IF Quantity = 0 THEN  
    EXIT;  
UnitPrice := TotalPrice / Quantity;
```

If the relational expression, *Quantity = 0*, evaluates to *TRUE*, the **EXIT** statement executes. This skips the rest of the code. If the relational expression evaluates to *FALSE*, the **EXIT** statement does not execute. Therefore, the next statement, which is the assignment statement, executes.



Note: This assignment statement is not indented, because it is not part of the **IF** statement.

Code Indentation

To indent several lines of code, select the lines, and then press TAB. The code indents two spaces every time that TAB is pressed. To indent several lines of code to the left, select those lines, and press SHIFT+TAB. This indents the code with one space to the left, every time that SHIFT+TAB is pressed. Block indentation enables developers to indent multiple lines of code in only three keystrokes (one for highlighting the code and two for pressing SHIFT and TAB).

The CASE Statement

Use the **CASE** conditional statement when there are more than two possible values for the condition.

The Syntax of the CASE Statement

The **CASE** statement has the following syntax:

Code Example

```
CASE <expression> OF
  <value set 1> : <statement 1>;
  <value set 2> : <statement 2>;
  <value set n> : <statement n>;
  [ELSE <statement n+1>]
END
```

A value set resembles the constant set that is used with the **IN** operator, except that the value set does not have brackets. Value set can contain any of the following:

- A single value
- Multiple values separated by commas
- Ranges of values
- Multiple ranges of values that are separated by commas

All values in the value sets must have a type comparable with the type of the expression. In most cases, the data type of the value sets are converted to the data type of the evaluated expression. The only exception is when the evaluated expression is a **Code** variable, then the value sets are not converted to the **Code** data type.

The **ELSE** clause is optional in the **CASE** statement.

When the **CASE** statement executes, the expression is evaluated first. This expression is known as the *selector*. Then, each value in each value set is evaluated in order.

If one of the values in the first value set matches the value of the expression, the first statement executes. If one of the values in the second value set matches the value of the expression, the second statement executes. This continues to the last value set.

If the last value set is reached and no statement executes, the **ELSE** clause is checked. If there is an **ELSE** clause, the statement following it, execute. If there is no **ELSE** clause, no statement executes for the **CASE** statement.

Only one of the statements executes. If there is more than one value set that contains the value that matches the expression, the statement following the earliest value executes.

Compound Statements and Comments

Understanding the concepts behind compound statements and comments helps to lay the groundwork for understanding nested **IF** statements.

Compound Statement

A compound statement is multiple statements. It improves the capabilities of the conditional statement. For example, in an **IF** statement, when a condition tests to *TRUE*, use a compound statement after the **THEN** to perform two or more assignments. Usually, after the **THEN**, only one statement executes. However, if that statement is a compound statement, multiple assignment statements can be performed in it.

Comment

A comment is a description that developers write in their code. The purpose is to explain the code, document who made a modification, or why the modification was made. Because a comment is a description, and not an actual segment of code, developers do not want the compiler to read it. Two slash marks (//) or a set of braces ({}) indicate to the compiler that these lines of code are comments, and should not be translated.

The Syntax of Compound Statements

Use the following syntax for compound statements:

Code Example

```
BEGIN <statement> {; <statement>} END
```

The braces ({}) indicate that whatever is included inside the brackets can be repeated zero or more times. In other words, there may not be a second statement, or there may be two or five, or as many as needed.

IF - THEN Compound Statement

Use a compound statement wherever, syntactically, you can use a single statement. For example, to calculate a unit price from an extended price and then add the extended price to the total price (only if the quantity is not zero[0]), use the following **IF** statement:

Code Example

```
IF Quantity <> 0 THEN BEGIN  
    UnitPrice := ExtendedPrice / Quantity;  
    TotalPrice := TotalPrice + ExtendedPrice;  
END;
```



Note: The statements in a compound statement are indented two spaces from the **BEGIN** and the **END**. The **BEGIN** and the **END** are aligned with one another.

If you follow this convention strictly, together with the general indentation conventions for **IF** statements, the result is as follows:

Code Example

```
IF Quantity <> 0 THEN  
  
    BEGIN  
  
        UnitPrice := ExtendedPrice / Quantity;  
  
        TotalPrice := TotalPrice + ExtendedPrice;  
  
    END;
```


To prevent this double indentation, but still make the code easy to read, there is a special indentation convention for **IF** statements. In an **IF** statement, the **BEGIN** is positioned on the same line as the **THEN**, and the **END** is aligned with the beginning of the line that contains the **BEGIN**.

IF - THEN - ELSE Compound Statement

Suppose that the *Quantity* is zero. To set the unit price to zero and skip the rest of the trigger, use the **ELSE** clause. The general indentation conventions look as follows:

Code Example

```
IF Quantity <> 0 THEN  
  
    BEGIN  
        UnitPrice := ExtendedPrice / Quantity;  
        TotalPrice := TotalPrice + ExtendedPrice;  
    END  
  
ELSE  
  
    BEGIN  
        UnitPrice := 0;  
        EXIT;  
    END;
```

There are special indentation conventions to use with **ELSE** clauses. To reduce the number of lines of code (this keeps more lines on the screen at one time), but still making the code easy to read, there are two changes from the previous code sample. First, the **ELSE** appears on the same line as the **END** from its corresponding **IF**. Second, the **BEGIN** appears on the same line as the **ELSE**. Therefore, the Microsoft Dynamics NAV standard indentation for the previous **IF** statement is as follows:

Code Example

```
IF Quantity <> 0 THEN BEGIN  
    UnitPrice := ExtendedPrice / Quantity;  
    TotalPrice := TotalPrice + ExtendedPrice;  
END ELSE BEGIN  
    UnitPrice := 0;  
    EXIT;  
END;
```

Compound Statement by Using Nested IF Statements

In a nested **IF** statement, the statement following the **THEN** or the **ELSE** clause of an **IF** statement is another **IF** statement, instead of simple assignments.

Nested IF Statements Rule

The rule for nested **IF** statements is that the **ELSE** clause matches the closest **IF** above it that does not have an **ELSE** clause. Unindented lines of code can be difficult to read and understand, especially with nested **IF** statements. For example, examine the following code:

Code Example

```
IF Amount <> 0 THEN
IF Amount > 0 THEN
Sales := Sales + Amount
ELSE
IF Reason = Reason::Return THEN
IF ReasonForReturn = ReasonForReturn::Defective THEN
Refund := Refund + Amount
ELSE
Credits := Credits + Amount
ELSE
Sales := Sales - Amount;
```

In the previous code sample, it is difficult to tell the circumstances in which *Sales* are to be reduced by the *Amount* without carefully diagnosing the code. Incorrect indentation can make it even more difficult.

Code Example

```
IF Amount <> 0 THEN
  IF Amount > 0 THEN
    Sales := Sales + Amount
  ELSE
    IF Reason = Reason::Return THEN
      IF ReasonForReturn = ReasonForReturn::Defective THEN
        Refund := Refund + Amount
      ELSE
        Credits := Credits + Amount
    ELSE
      Sales := Sales - Amount;
```

Correct Nested IF Statements Indentation

By using the rule that the **ELSE** clause matches the closest **IF** without an existing **ELSE** clause, and by using the standard indentation conventions, you can rewrite the code sample as follows:

Code Example

```
IF Amount <> 0 THEN
  IF Amount > 0 THEN
    Sales := Sales + Amount
  ELSE
    IF Reason = Reason::Return THEN
      IF ReasonForReturn = ReasonForReturn::Defective THEN
        Refund := Refund + Amount
      ELSE
        Credits := Credits + Amount
    ELSE
      Sales := Sales - Amount;
```

In all these code samples, the code executes the same way, because the compiler ignores all spaces and new lines. However, in the last code sample, it is much easier for the developer to determine the action.

Variable Naming

Another important concept is variable naming. Even if the same code is indented correctly, if the variables' names are not meaningful, it is difficult to understand what the code is meant to do. The following code sample shows poorly named variables:

Code Example

```
IF Amt1 <> 0 THEN
  IF Amt1 > 0 THEN
    Amt2 := Amt2 + Amt1
  ELSE
    IF OptA = 1 THEN
      IF OptB = 3 THEN
        Amt3 := Amt3 + Amt1
      ELSE
        Amt4 := Amt4 + Amt1
    ELSE
      Amt2 := Amt2 - Amt1;
```

The difference between poor programming and good programming frequently is in how well the code documents itself. Some good strategies for writing clear, organized code are as follows:

- Use correct indentation.
- Use meaningful variable names.
- Do not use negative variable names (for example, **DoNotDelete**).
- Use Booleans to designate Yes or No choices.
- Use the option type and option constants rather than integers to designate selections with more than two choices.

Follow these guidelines to create easy-to-read code. Provide additional documentation to the code, and remember to add comments to the code.

Demonstration: Use the Conditional and Compound Statements in a Page

The following demonstration shows how to use the **IF**, **EXIT**, **CASE**, and compound statements in a page.

Demonstration Steps

1. Create a new page, and then add variables.
 - a. Create a new blank page, and save it as page **90006, Test Statements Page**.
 - b. Click **View > C/AL Globals**, and define the following global variables:

| Name | DataType |
|--------------|----------|
| Quantity | Integer |
| UnitPrice | Decimal |
| TotalSales | Decimal |
| TotalCredits | Decimal |
| GrandTotal | Decimal |
| Result | Decimal |

- c. Close the **C/AL Globals** window.
- d. Compile and save the page.

2. Add controls to the page.
 - a. Type the following on the first line of the **Page Designer**, to specify the **ContentArea** container and name it **My Test Page 1**:

| Caption | Type | SubType |
|----------------------|-----------|-------------|
| Test Statements Page | Container | ContentArea |

- b. Go to the next line and type the following to add a FastTab named **General**. Make sure that it is indented under the container control.

| Caption | Type | SubType |
|---------|-------|---------|
| General | Group | Group |

- c. Go to the next line and type the following to add a group control with Caption **Input**. Make sure that it is indented under the **General** FastTab.

| Caption | Type | SubType |
|---------|-------|---------|
| Input | Group | Group |

- d. Go to the next line and type the following to add two field controls that have **SourceExpr Quantity** and **UnitPrice**. Make sure that they are indented under the **Input** group.

| Caption | Type | SourceExpr |
|------------|-------|------------|
| Quantity | Field | Quantity |
| Unit Price | Field | UnitPrice |

- e. Set the **MinValue** property for the **UnitPrice** field to **0**.
 - f. Go to the next line and type the following to add a group control with **Caption Output**. Indent it to the same level as the **Input** group.

| Caption | Type | SubType |
|---------|-------|---------|
| Output | Group | Group |

- g. Go to the next line and type the following to add the following four controls. Make sure that they are indented under the **Output** group:

| Caption | Type | SourceExpr |
|-------------|-------|------------|
| Result | Field | Result |
| Total Sales | Field | TotalSales |

| Caption | Type | SourceExpr |
|---------------|-------|--------------|
| Total Credits | Field | TotalCredits |
| Grand Total | Field | GrandTotal |

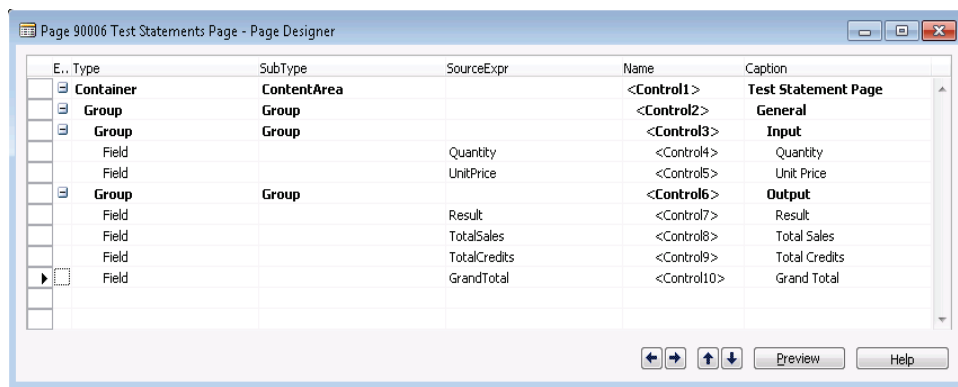


FIGURE 06.1: TEST STATEMENTS PAGE

- h. Set the **Editable** property for all four controls to *FALSE*.
 - i. Compile, and then save the page.
3. Use the IF and the EXIT statement.
- a. Open the **Action Designer** for the page.
 - b. Type the following on the first line of the **Action Designer** to add an **ActionContainer**.

| Type | SubType |
|-----------------|-------------|
| ActionContainer | ActionItems |

- c. Go to the next line and type the following to add an action. Make sure that it is indented under the **ActionItems ActionContainer**.

| Caption | Type |
|------------|--------|
| Execute IF | Action |

- d. Go to the next line and type the following to add another action. Make sure that it is indented under the **ActionItems ActionContainer**.

| Caption | Type |
|---------|--------|
| Clear | Action |

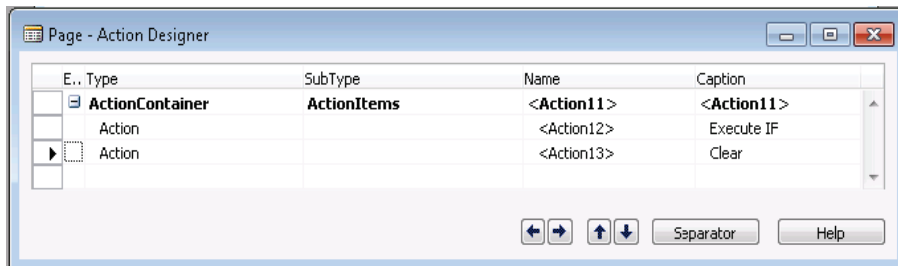


FIGURE 06.2: THE ACTION DESIGNER PAGE

- e. Click **View > C/AL Code** to open the **C/AL Editor**.
- f. Locate the **OnAction** trigger for the **Execute IF** action.
- g. Type the following code into the **OnAction** trigger of the action:

Code Example

```

IF Quantity = 0 THEN

    EXIT;

Result := Quantity * UnitPrice;

IF Result < 0 THEN

    TotalCredits := TotalCredits + Result

ELSE

    TotalSales := TotalSales + Result;

GrandTotal := GrandTotal + Result;
    
```

- h. Locate the **OnAction** trigger for the **Clear** action.
- i. Type the following code into the **OnAction** trigger of the action:

Code Example

```

Quantity := 0;

UnitPrice := 0;

Result := 0;

TotalSales := 0;

TotalCredits := 0;

GrandTotal := 0;
    
```

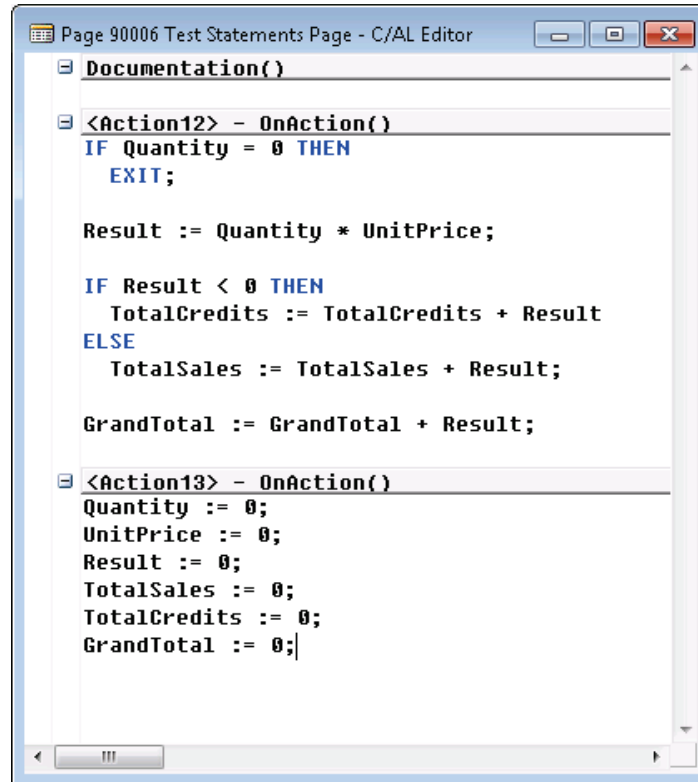


FIGURE 06.3: THE C/AL EDITOR – TEST STATEMENTS PAGE

- j. Close the **C/AL Editor** and close **Action Designer**.
 - k. Compile, save, and then close the page.
4. Test the page.
- a. Run page **90006, Test Statements Page**.
 - b. Type a value into the **Unit Price** and **Quantity** text box, and then click the **Execute IF** action. View the results.

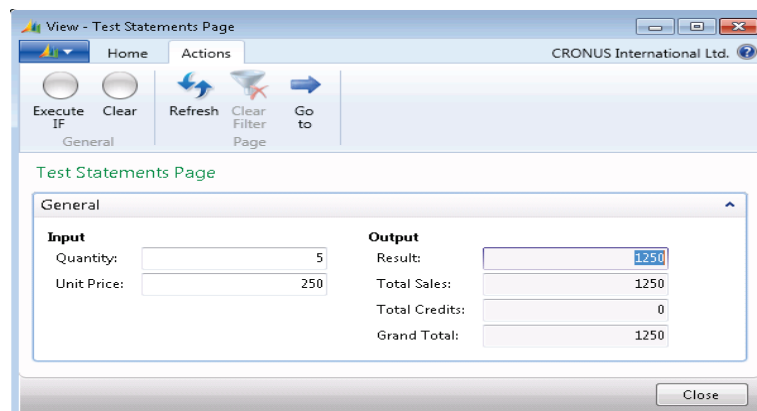


FIGURE 06.4: TEST STATEMENTS PAGE

- c. Type another value into the **Quantity** text box, and then click the **Execute IF** action again. View the results.
 - d. Try entering a negative value in the **Quantity** text box, (for example -4), and then click the **Execute** action. View the results.
 - e. Click the **Clear** action, and then see the results.
5. Use compound statements.
- a. Design page **90006, Test Statements Page**, from the Object Designer.
 - b. Click **View, C/AL Globals** and define the following global variables:

| Name | DataType |
|-----------------|----------|
| TotalQtySales | Integer |
| TotalQtyCredits | Integer |
| GrandQtyTotal | Integer |

- c. Close the **C/AL Globals** window.
- d. Go to the last line of the **Page Designer** and add the following into three field controls. Make sure that they are indented under the **Output** group.

| Caption | Type | SourceExpr |
|-------------------|-------|-----------------|
| Total Qty Sales | Field | TotalQtySales |
| Total Qty Credits | Field | TotalQtyCredits |
| Grand Qty Total | Field | GrandQtyTotal |

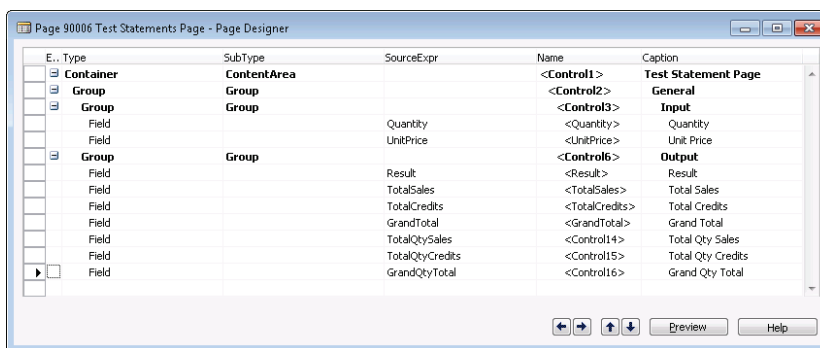


FIGURE 06.5: TEST STATEMENTS PAGE

- e. Set the **Editable** property for all three field controls to *FALSE*.
- f. Open the **Action Designer** for the page.

- g. Go to the next line and type the following to add an action. Make sure that it is indented under the **ActionItems ActionContainer**.

| Caption | Type |
|------------------|--------|
| Execute Compound | Action |

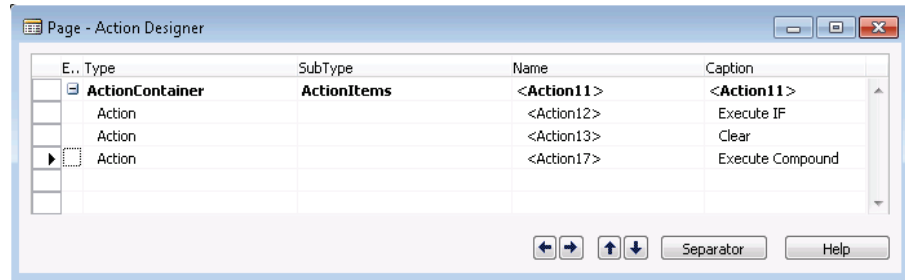


FIGURE 06.6: THE ACTION DESIGNER

- h. Click **View > C/AL Code** to open the **C/AL Editor**.
- i. Locate the **OnAction** trigger for the **Execute Compound** action.
- j. Type the following code into the **OnAction** trigger of the action:

Code Example

```

IF Quantity = 0 THEN
    EXIT;

Result := Quantity * UnitPrice;

IF Result < 0 THEN BEGIN
    TotalCredits := TotalCredits + Result;
    TotalQtyCredits := TotalQtyCredits + Quantity;
END ELSE BEGIN
    TotalSales := TotalSales + Result;
    TotalQtySales := TotalQtySales + Quantity;
END;

GrandTotal := GrandTotal + Result;
GrandQtyTotal := GrandQtyTotal + Quantity;
    
```

- k. Close the **C/AL Editor**, and then close the **Action Designer**.
- l. Compile, save, and then close the page.
- m. Run the page and test the changes.

6. Use the CASE statement.
 - a. Design page **90006, Test Statements Page**, from the **Object Designer**.
 - b. Open the **Action Designer** for the page.
 - c. Go to the next line and type the following to add an action. Make sure that it is indented under the **ActionItems ActionContainer**.

| Caption | Type |
|--------------|--------|
| Execute CASE | Action |

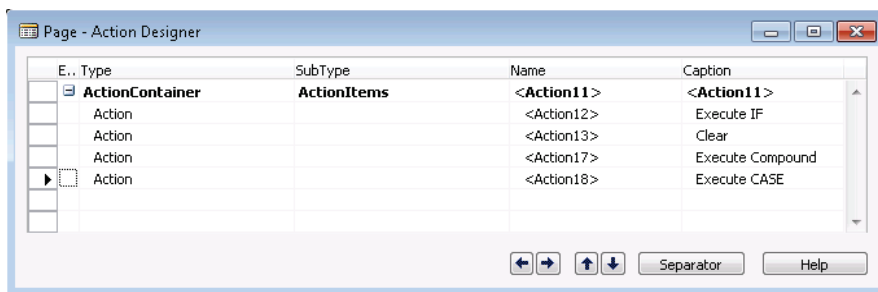


FIGURE 06.7: THE ACTION DESIGNER

- d. Click **View > C/AL Code** to open the **C/AL Editor**.
- e. Locate the **OnAction** trigger for the **Execute CASE** action.
- f. Type the following code into the **OnAction** trigger of the action:

Code Example

```

Result := Quantity * UnitPrice;

CASE TRUE OF
Quantity = 0:
    EXIT;
Quantity < 0:
    BEGIN
        TotalCredits := TotalCredits + Result;
        TotalQtyCredits := TotalQtyCredits + Quantity;
    END;
Quantity > 0:
    BEGIN
        TotalSales := TotalSales + Result;
        TotalQtySales := TotalQtySales + Quantity;
    END;
END;

GrandTotal := GrandTotal + Result;
GrandQtyTotal := GrandQtyTotal + Quantity;
    
```

- g. Close the **C/AL Editor**, and then close the **Action Designer**.
- h. Compile, save, and then close the page.
- i. Run the page and test the changes.

The Syntax of Comments

There are two ways to write comments.

Single-Line Comment

Use a single-line comment to comment out a single line in the code. To comment out means to block code so that it is ignored by the compiler. You create a comment by putting two consecutive slashes (//) on a line. Everything after those two slashes is considered a comment and is ignored by the compiler. The following code sample shows single-line comments:

Code Example

```
// Calculate the Unit Price
IF Quantity <> 0 THEN // Do not allow division by zero
    UnitPrice := TotalPrice / Quantity
ELSE // that is, if Quantity = 0
    UnitPrice := 0;
```

The first comment is an example of a code line used completely for a comment. The other two comments (the second and fourth lines) show how to have code on a line that is followed with a comment. In each case, the compiler ignores anything following the two slashes on that line.

Block of Comments

Use a block of comments to comment out a block of code. If an opening brace ({} is inserted in the code, the compiler ignores everything that follows until the closing brace (}). The compiler also ignores new lines in this case. The following code sample shows a block of comments:

Code Example

```
{The following code is used to calculate the Unit Price. It compares the Quantity to zero in order to prevent division by zero. In this case, the code sets the Unit Price to zero.}

IF Quantity <> 0 THEN
    UnitPrice := TotalPrice / Quantity
ELSE
    UnitPrice := 0;
```

Nested Comments

Developers use a block of comments to track problems. An opening brace is inserted at the beginning of a section of code, and a matching closing brace is inserted at the end of the section of code. This comments out the whole section of code. This enables developers to focus on the remaining parts of the code, or eliminate a part of the code as the cause of a problem.

You can nest a block of comments. When the compiler reaches an opening brace, it treats everything as a comment until it reaches the matching closing brace. Therefore, in a trigger code, a closing brace must match every opening brace. The following code sample shows a nested block of comments:

Code Example

```
{The following code temporarily removed on 8/15/12...

{The following code is used to calculate the Unit Price. It compares the Quantity to zero to prevent division by zero. In this case, the code sets the Unit Price to zero.}

IF Quantity <> 0 THEN
    UnitPrice := TotalPrice / Quantity
ELSE
}
    UnitPrice := 0;
```

When the code in the example runs, only the line that sets the unit price to zero (the last line) actually executes. The other lines are skipped.

Practice: Nested IF

The questions on this self-test relate to the following code. Hand-execute this code to determine the answers.

The IF and ELSE Statement

Draw a line from each **ELSE** to its matching **IF** statement for the following set of statements.

Code Example

```
IF (X1 > X2) OR B3 THEN BEGIN IF X7 < X2 THEN  
  
A1 := (X1 + X7) / 2 ELSE IF X6 < X2 THEN  
  
A1 := (X1 + X6) / 2;  
  
X7 := X6 + 1; END ELSE IF (X1 < X2) AND B5 THEN  
  
IF X6 > X7 THEN BEGIN  
  
IF B2 THEN A2 := X1 / 2 + X7 / 2 ELSE  
  
A2 := X1 * 2 + X7 * 2; END ELSE A1 := X1 ELSE  
  
A2 := X2;  
  
IF B1 THEN EXIT;
```



Note: *HINT: It may be more efficient to rewrite the set of statements from the top to the bottom with correct indentation.*

After the following set of statements executes, what is the value of variable **A5**?

Code Example

```
{Initialize Variables}  
  
A5 := 7; // Initialize answer  
  
B1 := {TRUE;} FALSE; B2 := TRUE; // FALSE;  
  
A1 := { 5 // be sure to set this one correctly  
  
A2 := 3 * A5;  
  
A3 := 10 } 11; // either one is OK  
  
A2 := 2 * A5;  
  
// IF B2 THEN A5 := A5 + 1  
  
IF (A1 < A5) OR {B2 AND} B1 THEN  
  
A5 := 3 * A5 // ELSE A5 := A2 / A5;  
  
ELSE A5 := A1 + A5;
```

Lab 6.1: Use Conditional and Compound Statements

Scenario

Simon is a developer working for CRONUS International Ltd. CRONUS International has decided to start selling Microsoft Dynamics NAV training courses as its business.

Simon has already created a **Course** table to store course information, a **Course Card** page, and a **Course List** page to enter and display course information. Now Simon must add several extension fields to make the list page easier to use.

Exercise 1: Use Conditional and Compound Statements

Exercise Scenario

The **Course List** page must have three extension fields: **Level**, **Suggestion**, and **SPA** (Solution Provider Agreement). The **Level** and **Suggestion** fields are string fields, whereas the **SPA** field must be displayed as a check box. Depending on the **Difficulty** of the Course, the **Level** and **Suggestion** fields vary. Also, depending on the **Passing Rate** and the **Difficulty** of the Course, the **SPA** varies.

Task 1: Create global variables

High Level Steps

1. Add three global variables for Level, Suggestion, and SPA in the **Course List** page together with their respective data.
2. Add three field controls to the **Course List** page.
3. Add code to the **OnAfterGetRecord** trigger of the page.

Detailed Steps

1. Add three global variables for Level, Suggestion, and SPA in the **Course List** page together with their respective data.
 - a. Design page **90011, Course List**, from the **Object Designer**.
 - b. Click **View > C/AL Globals**, and define the following global variables:

| Name | DataType | Length |
|------------|----------|--------|
| Level | Text | 30 |
| Suggestion | Text | 80 |
| SPA | Boolean | |

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

2. Add three field controls to the **Course List** page.
 - a. Go to the last line of the **Page Designer** and type the following to add three field controls that have **SourceExpr Level**, **Suggestion**, and **SPA**. Make sure that these field controls are indented at the same level as the other field controls.

| Caption | Type | SourceExpr |
|------------|-------|------------|
| Level | Field | Level |
| Suggestion | Field | Suggestion |
| SPA | Field | SPA |

3. Add code to the **OnAfterGetRecord** trigger of the page.
 - a. Click **View > C/AL Code** to open the **C/AL Editor**.
 - b. Locate the **OnAfterGetRecord** trigger of the page.
 - c. Add the following code:

Code Example

```
Level := '';
Suggestion := '';
SPA := FALSE;
CASE Difficulty OF
  1..5:
    BEGIN
      Level := 'Beginner';
      Suggestion := 'Take e-Learning or remote training';
    END;
  6..8:
    BEGIN
      Level := 'Intermediate';
      Suggestion := 'Attend instructor-Led';
    END;
  9..10:
```



```
BEGIN  
  
    Level := 'Advanced';  
  
    Suggestion := 'Attend instructor-Led and self study';  
  
END;  
  
END;  
  
IF ("Passing Rate" >= 70) AND (Difficulty >= 6) THEN  
  
    SPA := TRUE;
```

- d. Close the **C/AL Editor**.
- e. Compile, save, and close the page.

Arrays

Arrays are complex variables that contain a group of variables with the same data type. They are special variables that have more functionality than the variables that were discussed earlier.

Data Types and Variables

Simple data types only have a single value. A simple variable is defined by using a simple data type. A *complex data type* has multiple values. A *complex variable* has a complex data type. That is, a variable that has multiple values.

Array

An *array* is a complex variable that holds a group of variables. This entire group is defined at the same time with a single identifier and a single data type. For example, a developer can create an array variable with the following identifier and data type:

- Identifier: QuantityArray
- Data Type: Integer

Element

An *element* is a single variable in an array. An array consists of one or more elements. You can create an array with one element, but it is easier to create a simple variable. All elements in an array have the same data type as the array. The previous array must have a defined number of elements. Following is an example:

- Elements: 5

Index

An *index* is used to refer to a single element in an array. To access a single element in an array, use both the array variable name and the index (this is a number) to indicate the desired array element. To access the fourth element in the previous array, use the following:

- QuantityArray[4]

Dimension

An array can have one or more dimensions. The simplest array is a one-dimensional array. By default, this is defined in the definition of arrays. This array only has elements in one dimension. This is the same as having elements only on the x-axis of a graph.

In the earlier example, there are a total of five elements. They consist of one line, or a dimension, of five defined elements.

If the array is defined as having two dimensions, it has a total of 25 elements (five elements in one dimension multiplied by five elements in the second dimension). This is the same as having a two-dimensional graph that uses the x-axis and y-axis. Each line has five elements (one through five), but provides a combination of 25 different points by using only integers.

Arrays can have up to 10 **Dimensions** in C/AL.

To use a particular element in a multidimensional array, specify an index value for each dimension. Therefore, there is the same number of index values as the defined dimensions. These index values must be integers.

The Syntax of Arrays

The difference between calling a simple variable and an array is the addition of the element index.

Variable Syntax

Use the variable identifier when you refer to any variable or array as a whole. For example, **CLEAR**, a built-in function, has the following syntax:

Code Example

```
CLEAR(<variable>)
```

This function clears the variable that is used as the parameter. If the variable is a numeric type, the **CLEAR** function sets it to zero. If the variable is a string type, the **CLEAR** function sets it to an empty string. If the variable is an array, the **CLEAR** function sets each element in the array to its own cleared value.

For example, for a one-dimensional array named **SaleAmount** of type **Decimal**, use the following function call to set all elements in **SaleAmount** to zero:

Code Example

```
CLEAR(SaleAmount);
```

Array Element Syntax

Refer to a single element of an array by using its identifier and its index, according to the following syntax:

Code Example

```
<identifier>[<index expression>{,<index expression>}]
```

The brackets ([]) are literal brackets, whereas the braces ({}), indicate that whatever is included inside the braces can be repeated zero or more times. A one-dimensional array requires a single index expression, whereas a two-dimensional array requires two index expressions, separated by a comma. Each index expression must result in an integer value when evaluated. To set the fifth element of the **SaleAmount** array to zero, use the following assignment statement:

Code Example

```
SaleAmount[5] := 0;
```

How to think about arrays

Think about a one-dimensional array as the following row of boxes:

| | | | | | | | |
|--------|--------|--------|--------|--------|--------|--------|--------|
| 5 | 27 | 8 | 17 | 25 | 3 | 7 | 12 |
| Box[1] | Box[2] | Box[3] | Box[4] | Box[5] | Box[6] | Box[7] | Box[8] |

FIGURE 06.8: ONE-DIMENSIONAL ARRAY ILLUSTRATION

This is a one-dimensional array, with the identifier *Box* and eight elements. Each element of the array is a single box. The fourth element of the array contains the value of *17*. To refer to the fourth element of the array, use *Box[4]*.

You can think of a two-dimensional array as the following checkerboard or a multiplication table:

| | | | | | | |
|----------|----------|----------|----------|----------|----------|----------|
| Box[1,c] | 1 | 2 | 3 | 4 | 5 | 6 |
| Box[2,c] | 2 | 4 | 6 | 8 | 10 | 12 |
| Box[3,c] | 3 | 6 | 9 | 12 | 15 | 18 |
| Box[4,c] | 4 | 8 | 12 | 16 | 20 | 24 |
| Box[5,c] | 5 | 10 | 15 | 20 | 25 | 30 |
| | Box[r,1] | Box[r,2] | Box[r,3] | Box[r,4] | Box[r,5] | Box[r,6] |

FIGURE 06.9: TWO-DIMENSIONAL ARRAY ILLUSTRATION

The example is a two-dimensional array with the identifier *Box*, and a total of 30 elements, broken up into five rows of six elements. Think of the first dimension as the row number, and the second dimension as the column number. To identify a specific element, developers must identify both the row and the column. For example, to have the value of the element in Row 4, Column 6, refer to *Box[4,6]*; the value is 24.

Three or more dimensions are more difficult to visualize. Think of the index as a list of criteria. There are few reasons to have more than one one-dimensional array in Microsoft Dynamics NAV 2013; however, you can have up to ten.

The Power of Arrays

You can use arrays to do more than access a particular element, such as the following:

- *SaleAmount[5]*

Creating an array with 10 elements only saves the developer from creating ten different variables. Arrays can do much more.

Expressions as an Index Value

The power of an array is that the index value can be an expression, such as the following:

- *SaleAmount[Counter]*

This expression enables the element to be determined at run time. In this example, the *Counter* value determines the element.

However, at run time, a developer might not know whether the value of the index expression is a valid element number or not. If an array is defined to have eight elements, and the code refers to element number 12, this results in a run-time error.

To avoid this error, you must use an **IF** statement to test the value of the variable before you use it to index an array. See the following example.

Code Example

```
IF Counter <= 8 THEN  
    SaleAmount[Counter] := Answer;
```

ARRAYLEN Function

The problem with this method is that the array actually might be defined to have seven elements and the problem might be identified only at run time. C/AL addresses this problem with a built-in function known as **ARRAYLEN** that has the following syntax:

Code Example

```
Result := ARRAYLEN(<array variable>)
```

The code in this example indicates that the **ARRAYLEN** function results in a value, in this case of type Integer. By using the previous syntax, the result is the number of elements in the array variable that is used as the parameter. For example, if a one-dimensional array is defined to have eight elements, the following function has the integer value of 8:

Code Example

```
ARRAYLEN(SaleAmount)
```



Note: *ARRAYLEN can also be used for multidimensional arrays.*

The advantage of using **ARRAYLEN** is that changes to the defined length of an array do not require developers to go back through their code to update references to the defined length. Therefore, in the first **ARRAYLEN** example, developers can test to see whether an index value is valid as follows:

Code Example

```
IF Counter <= ARRAYLEN(SaleAmount) THEN  
  
    SaleAmount[Counter] := Answer;
```

Because the number 8 is never used, the array does not have to have exactly eight elements. If the array is actually defined to have seven elements, this code still works perfectly.

Strings as Arrays of Characters

A *string variable*, or variable of type Text or Code, can be thought of as an array of characters. Therefore, C/AL allows for access to each character as an element in an array.

Characters of Elements

Each element is considered a variable of type **Char**. For example, running the following code results in the message "Walk in the dark":

Code Example

```
Str := 'Walk in the park';  
  
Str[13] := 'd';  
  
MESSAGE(Str);
```

Because Char is a numeric type, ASCII codes can be used for characters when you use strings in this manner.

For example, when importing a text line, tab characters (ASCII code 9) can be checked as follows:

Code Example

```
IF Str[idx] = 9 THEN  
  
    MESSAGE('There is a TAB character at position %1 in the text.',idx);
```

You cannot access the length of a string in this manner. Regardless of which characters are read or set, the length of the string remains the same. The elements beyond the string's length are considered undefined.

In the first string example, if the 25th element is set (instead of the 13th) to "d", because the 25th character is beyond the length of the string, this causes a run-time error.

To know the number of elements in a string that are used as an array of characters, use the **STRLEN** function, instead of the **ARRAYLEN** function.



Note: Do not use **MAXSTRLEN** when accessing strings as arrays because it returns the declared length of a string. Always use **STRLEN** because it returns the actual length of a string. If you try to assign a value to an element beyond the actual length of a string, even if the declared length is larger, a run-time error occurs.

Repetitive Statements

A *repetitive statement* is a statement that enables execution of one or more statements multiple times. There are several types of repetitive statements. The difference between repetitive statements is in the number of times these statements are executed and how that number is determined. A repetitive statement is known as a *loop*, because when the execution reaches the end of the repetitive statement, it loops back to the beginning of the statement.

The FOR Statement

Use the **FOR** statement when a statement is executed a predetermined number of times.

The FOR...TO Statement

The **FOR...TO** statement has the following syntax:

Code Example

```
FOR <control variable> := <start value> TO <end value> DO <statement>
```

The control variable must be a variable of type Boolean, Date, Time, or any numeric type. The start value and the end value must be either expressions that evaluate to the same data type as the control variable, or variables of the same data type as the control variable. The following code sample shows a **FOR...TO** statement:

Code Example

```
FOR idx := 4 TO 8 DO
    Total := Total + 2.5;
```

In this case, the control variable (*idx*), the start value (4), and the end value (8) are all integers. The following steps describe the process:

| Step | Code |
|---|----------------|
| 1. The start value expression is evaluated and the control variable is set to the result. | idx := 4 |
| 2. The end value expression (after the TO) is evaluated. | End value is 8 |

| Step | Code |
|---|--|
| 3. The control variable is compared to the end value. If it is greater than the end value, the FOR statement is ended. | IF idx > 8, THEN The FOR statement ends. |
| 4. The statement (after the DO) is executed. | Total := Total + 2.5 |
| 5. The control variable is incremented by one. | idx := idx + 1 (5) |
| 6. Go to step 3 and test the control variable again. | |

In this example, the *Total* variable was increased by two and one-half for five times. Therefore, it is increased by 12.5. Both the start value and the end value are evaluated once at the beginning of the **FOR** statement.

The control variable must not be changed in the **FOR** loop; if it is changed, the result is not predictable. The control variable value outside the **FOR** loop (after it ends) is not defined.

If several statements must run inside the loop, create a compound statement by adding **BEGIN** and **END**. The following code sample shows how to execute two statements in a loop:

Code Example

```
FOR idx := 4 TO 8 DO BEGIN

    Total := Total + 2.5;

    GrandTotal := GrandTotal + Total;

END;
```

The FOR...DOWNTON Statement

The **FOR...DOWNTON** statement has the following syntax:

Code Example

```
FOR <control variable> := <start value> DOWNTON <end value> DO <statement>
```


The rules for the control variable, start value, and end value are the same as for the **FOR...TO** statement. The only difference between the two is that in the **FOR...TO** statement, the control variable increases in value until it is greater than the end value, whereas in the **FOR.. DOWNTO** statement, the control variable decreases in value until it is less than the end value. The following code sample shows a **FOR...DOWNTO** statement with an array and a compound statement:

Code Example

```
FOR idx := 9 DOWNTO 1 DO BEGIN

    TotalSales := TotalSales + Sales[idx];

    NumberSales := NumberSales + 1;

END;
```

The following steps describe the process:

| Step | Code |
|--|--|
| 1. The start value expression is evaluated and the control variable is set to the result. | idx := 9 |
| 2. The end value expression (after the DOWNTO) is evaluated. | End value is 1 |
| 3. The control variable is compared to the end value. If it is less than the end value, the FOR statement is ended. | IF idx < 1, THEN The FOR statement ends. |
| 4. The statement (after the DO) is executed. | TotalSales := TotalSales + Sales[9]; NumberSales := NumberSales + 1; |
| 5. The control variable is decreased by one. | idx := idx - 1 (8) |
| 6. Go to step 3 and test the control variable again. | |

Through each execution of the first statement in the compound statement, a different element of the *Sales* array is accessed: first the ninth, and then the eighth, and so on.

The WHILE...DO Statement

Use the **WHILE** loop when a statement is to be executed, as long as some condition is true. a **WHILE** loop has the following syntax:

Code Example

```
WHILE <Boolean expression> DO <statement>
```

The **WHILE...DO** statement is simpler than the **FOR** statement. As long as the Boolean expression evaluates to *TRUE*, the statement is executed repeatedly. As soon as the Boolean expression evaluates to *FALSE*, the statement is skipped and the execution continues with the statement following it. The following code sample shows a **WHILE...DO** statement with a compound statement:

Code Example

```
WHILE Sales[idx + 1] <> 0 DO BEGIN
    Idx := idx + 1;
    TotalSales := TotalSales + Sales[idx];
END;
```

The following steps describe the process:

| Step | Code |
|---|---|
| 1. The Boolean expression is evaluated. If it evaluates to <i>TRUE</i> , go to step 2. If it does not evaluate, the WHILE statement is ended. | Is Sales[idx + 1] <> 0? |
| 2. The statement (after the DO) is executed. | idx := idx + 1; TotalSales := TotalSales + Sales[idx]; |
| 3. Go to step 1 and test the Boolean expression again. | |

The Boolean expression is tested before the statement is executed even one time. If it evaluates to *FALSE* from the beginning, the statement is never executed.

The *Sales[idx]* that is added to *TotalSales* in the **WHILE** loop is the same value that was tested in the *Sales[idx + 1]* at the beginning of the **WHILE** loop. The intervening *idx := idx + 1* statement causes this. As soon as the **WHILE** loop has ended, **idx** still refers to the last nonzero element in the **Sales** array.

Unlike in **FOR** statements, it is the developer's responsibility to make sure that the expression evaluates to *FALSE* at some point. This prevents a never-ending loop.

The REPEAT...UNTIL Statement

Use the **REPEAT** statement when one or more statements are to be executed until some condition becomes true. It has the following syntax:

Code Example

```
REPEAT <statement> { ; <statement> } UNTIL <Boolean expression>
```

Following are several differences between the **REPEAT** and **WHILE** statements:

- There can be more than one statement between the **REPEAT** and the **UNTIL**, even if no **BEGIN** and **END** is used.
- The Boolean expression is not evaluated until the end, after the statements are already executed once.
- When the Boolean expression is evaluated, it loops back to the beginning if the expression evaluates to *FALSE*. If the expression evaluates to *TRUE*, it ends the loop.

The following code sample shows a **REPEAT...UNTIL** statement that achieves the same result with the **WHILE** statement code sample:

Code Example

```
REPEAT
    Idx := idx + 1;
    TotalSales := TotalSales + Sales[idx];
UNTIL Sales[idx] = 0;
```

The following steps describe the process:

| Step | Code |
|--|---|
| 1. The statements (between the REPEAT and the UNTIL) are executed. | idx := idx + 1; TotalSales := TotalSales + Sales[idx]; |
| 2. The Boolean expression is evaluated. If it evaluates to <i>TRUE</i> , the REPEAT statement is ended. If it does not evaluate, go back to step 1. | Is Sales[idx] = 0? |

Because the Boolean expression is not evaluated until the end, the statements incrementing the index and adding the *TotalSales* are executed, even though the value of those *Sales* might be zero. Therefore, at the end of this loop, *idx* refers to

the first *Sales* which equals zero, instead of the last nonzero *Sales* as in the **WHILE** statement.

The Boolean expression had to be rewritten, because a *FALSE* condition is required to continue the loop, instead of a *TRUE* condition as in the **WHILE** statement.

Just as in the **WHILE...DO** statement, it is the developer's responsibility to make sure that the Boolean expression evaluates to *TRUE* at some point to prevent a never-ending loop.

Demonstration: Use Arrays and Repetitive Statements

The following demonstration shows how to use arrays and repetitive statements in a page. The objective is to create a page that takes 10 input numbers from the user, store them in arrays, and sorts them by using several repetitive statements.

Demonstration Steps

1. Create a new page and add variables.
 - a. Create a new blank page, and save it as page **90007, Test Array Page**.
 - b. Click **View > C/AL Globals**, and define the following global variables:

| Name | Data Type | Purpose |
|--------------|-----------|---|
| InputNumber | Integer | Use this to store the input numbers. |
| OutputNumber | Integer | Use this to store the output numbers. |
| LoopCount | Integer | Use this to count how many loops are required for the sorting routine. |
| SwapCount | Integer | Use this to count how many swaps actually occur in the sorting routine. |
| idx | Integer | Use this as an expression for the array's index. |
| IsSorted | Boolean | Use this as a flag for whether a repetitive statement must be repeated. |
| LowestSwitch | Integer | Use this to improve the sorting routine. |
| TempNumber | Integer | Use this for temporary holding values to be swapped. |

- c. Open the **Properties** window for the *InputNumber* variable, and then set the following property:
 - d. **Dimensions:** 10



Note: This means that `InputNumber` is a one-dimensional array variable of type `Integer`, with a maximum of elements of the first (and only) dimension, 10, which means that there are 10 elements.

To create a two-dimensional array, put two numbers in the **Dimensions** property that are separated by a semicolon (;). For example, to create a two-dimensional array in which the maximum number of elements of the first dimension is five, and the maximum number of elements of the second dimension is seven, put 5;7 in the **Dimensions** property. This means that there are 35 elements.

- e. Do the same for the **OutputNumber** variable.
 - f. Close the **Properties** window, and then close the **C/AL Globals** window.
 - g. Compile and save the page.
2. Add controls to the page.
- a. Type the following on the first line of the **Page Designer**, to specify the **ContentArea** container and name it **Test Array Page**:

| Caption | Type | SubType |
|-----------------|-----------|-------------|
| Test Array Page | Container | ContentArea |

- b. Go to the next line and type the following to add a **FastTab** named **General**. Make sure that it is indented under the container control.

| Caption | Type | SubType |
|---------|-------|---------|
| General | Group | Group |

- c. Go to the next line and type the following to add a group control with **Caption Input**. Make sure that it is indented under the **General FastTab**.

| Caption | Type | SubType |
|---------|-------|---------|
| Input | Group | Group |

- d. Go to the next line and type the following to add 10 field controls with **SourceExpr InputNumber** and its element index. Make sure that the field controls are indented under the **Input** group.

| Caption | Type | SourceExpr |
|----------------|-------|----------------|
| InputNumber[1] | Field | InputNumber[1] |

| Caption | Type | SourceExpr |
|-----------------|-------|-----------------|
| InputNumber[2] | Field | InputNumber[2] |
| InputNumber[3] | Field | InputNumber[3] |
| InputNumber[4] | Field | InputNumber[4] |
| InputNumber[5] | Field | InputNumber[5] |
| InputNumber[6] | Field | InputNumber[6] |
| InputNumber[7] | Field | InputNumber[7] |
| InputNumber[8] | Field | InputNumber[8] |
| InputNumber[9] | Field | InputNumber[9] |
| InputNumber[10] | Field | InputNumber[10] |



Note: This creates the input column in the page

- e. Go to the next line and type the following to add a group control with **Caption Output**. Indent the group control to the same level as the **Input** group.

| Caption | Type | SubType |
|---------|-------|---------|
| Output | Group | Group |

- f. Go to the next line and type the following to add 10 field controls with **SourceExpr OutputNumber** and its element index. Make sure that the field controls are indented under the **Output** group.

| Caption | Type | SourceExpr |
|------------------|-------|------------------|
| OutputNumber[1] | Field | OutputNumber[1] |
| OutputNumber[2] | Field | OutputNumber[2] |
| OutputNumber[3] | Field | OutputNumber[3] |
| OutputNumber[4] | Field | OutputNumber[4] |
| OutputNumber[5] | Field | OutputNumber[5] |
| OutputNumber[6] | Field | OutputNumber[6] |
| OutputNumber[7] | Field | OutputNumber[7] |
| OutputNumber[8] | Field | OutputNumber[8] |
| OutputNumber[9] | Field | OutputNumber[9] |
| OutputNumber[10] | Field | OutputNumber[10] |



Note: This creates the output column in the page.

- g. Set the **Editable** property for all ten field controls to *FALSE*.
- h. Go to the next line and type the following to add a group control with **Caption Counter**. Indent it to the same level as the **Input** group.

| Caption | Type | SubType |
|---------|-------|---------|
| Counter | Group | Group |

- i. Go to the next line and type the following to add two field controls with **SourceExpr** *LoopCount* and *SwapCount*. Make sure that it is indented under the **Counter** group.

| Caption | Type | SubType |
|------------|-------|-----------|
| Loop Count | Field | LoopCount |
| Swap Count | Field | SwapCount |



Note: This creates a section in the page to show the performance of the sorting routine. The less loop count, the better the sorting routine.

- j. Set the **Editable** property to *FALSE* for the two counter field controls *FALSE*.

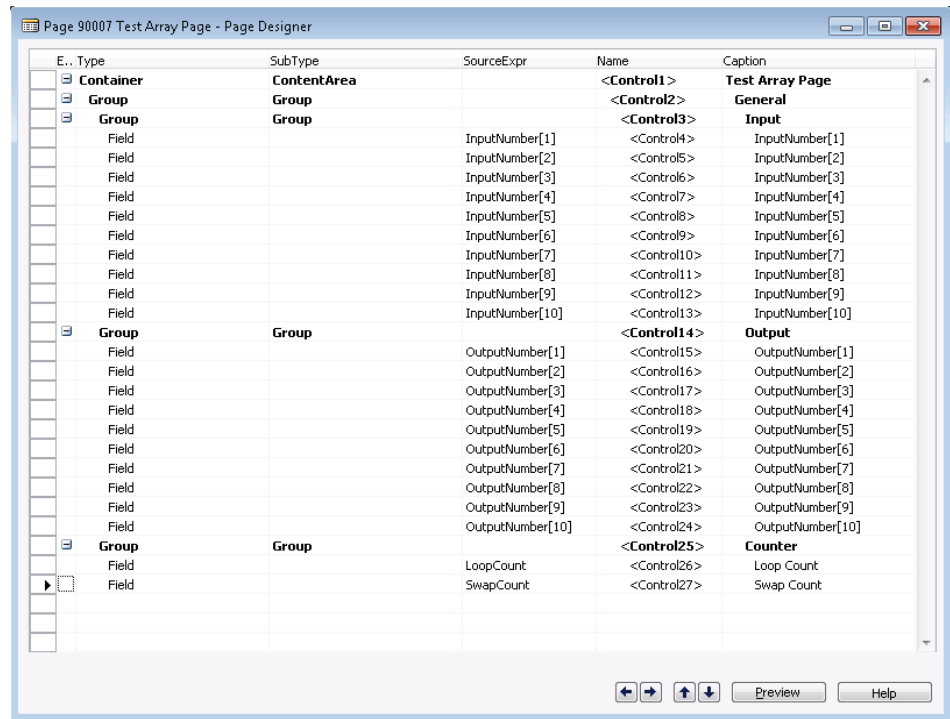


FIGURE 06.10: TEST ARRAY PAGE

- k. Compile and save the page.
3. Add code to clear the page.
 - a. Open the **Action Designer** for the page.
 - b. Type the following on the first line of the **Action Designer** to add an **ActionContainer**.

| Type | SubType |
|-----------------|-------------|
| ActionContainer | ActionItems |


- c. Go to the next line and type the following to add an action. Make sure that it is indented under the **ActionItems ActionContainer**.

| Caption | Type |
|---------|--------|
| Clear | Action |

- d. Click **View > C/AL Code** to open the **C/AL Editor**.
- e. Locate the **OnAction** trigger for the **Clear** action.
- f. Type the following code into the **OnAction** trigger of the action:

Code Example

```
CLEAR(InputNumber);
CLEAR(OutputNumber);
LoopCount := 0;
SwapCount := 0;
```

 **Note:** This adds an action to clear the page, by clearing the arrays and setting all the variables that are displayed in the page to 0.

- g. Close the **C/AL Editor**.
4. Add code to generate input.
- a. In the **Action Designer**, go to the next line and type the following to add another action. Make sure that it is indented under the **ActionItems ActionContainer**.


| Caption | Type |
|----------------|--------|
| Generate Input | Action |

- b. Click **View > C/AL Code** to open the **C/AL Editor**.
- c. Locate the **OnAction** trigger for the **Generate Input** action.
- d. Type the following code into the **OnAction** trigger of the action:

Code Example

```
LoopCount := 0;
SwapCount := 0;

FOR idx := 1 TO ARRAYLEN(InputNumber) DO
    InputNumber[idx] := RANDOM(ARRAYLEN(InputNumber));
```

 **Note:** A built-in C/AL function, **RANDOM**, achieves this result. A **BEGIN** is not required after the **DO** in the **FOR** statement because there is only one statement following the **FOR** statement.

- e. Close the **C/AL Editor**.

5. Add code to populate output.
 - a. In the **Action Designer**, go to the next line and type the following to add another action. Make sure that the action is indented under the **ActionItems ActionContainer**.

| Caption | Type |
|-----------------|--------|
| Populate Output | Action |

- b. Click **View > C/AL Code** to open the **C/AL Editor**.
 - c. Locate the **OnAction** trigger for the **Populate Output** action.
 - d. Type the following code into the **OnAction** trigger of the action:

Code Example

```

LoopCount := 0;
SwapCount := 0;

idx := 1;
WHILE (idx <= ARRAYLEN(InputNumber)) AND (idx <=
ARRAYLEN(OutputNumber)) DO BEGIN
    OutputNumber[idx] := InputNumber[idx];
    idx += 1;
END;
    
```

- e. Close the **C/AL Editor**.
6. Add code to sort the output.
 - a. In the **Action Designer**, go to the next line and type the following to add another action. Make sure that the action is indented under the **ActionItems ActionContainer**.

| Caption | Type |
|---------|--------|
| Sort | Action |

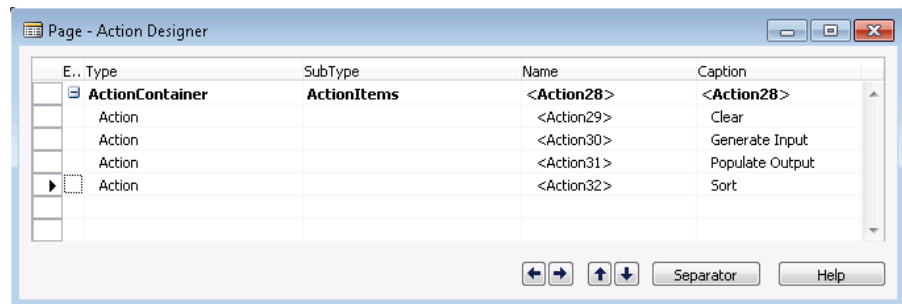



FIGURE 06.11: THE ACTION DESIGNER

- b. Click **View > C/AL Code** to open the **C/AL Editor**.
- c. Locate the **OnAction** trigger for the **Sort** action.
- d. Type the following code into the **OnAction** trigger of the action:


Code Example

```
LoopCount := 0;
SwapCount := 0;

REPEAT
  IsSorted := TRUE;
  FOR idx := ARRAYLEN(OutputNumber) DOWNTO 2 DO BEGIN
    LoopCount += 1;
    IF OutputNumber[idx] < OutputNumber[idx-1] THEN BEGIN
      TempNumber := OutputNumber[idx];
      OutputNumber[idx] := OutputNumber[idx-1];
      OutputNumber[idx-1] := TempNumber;
      SwapCount += 1;
      IsSorted := FALSE;
    END;
  END;
UNTIL IsSorted
```

 **Note:** First, the code initializes the two variables, LoopCount and SwapCount, to zero. Then the **REPEAT...UNTIL** statement loops until the output column does not require sorting any longer. The **FOR...DOWNTO** statement inside the **REPEAT...UNTIL** statement loops the output column to sort from the last element to the first element.

- e. Close the **C/AL Editor**.

 **Note:** The **FOR** statement only goes to two, instead of one because when idx is one, this causes it to refer to an array element of zero.

The three lines of the compound statement (between **BEGIN** and **END**) in the **FOR** loop does the actual swapping between elements of the array.

The IsSorted variable signals that sorting is finished. Set the IsSorted variable to TRUE at the beginning of each **REPEAT** loop. If you have to swap two elements because the list is not in order, set it to FALSE. At the end of the **REPEAT** loop, check whether it is TRUE. If it is, all the elements are in order, because no elements needed to be swapped. Now exit the loop. If elements need to be swapped, the loop is repeated again. A Boolean variable that is used as a signal is known as a flag.

You cannot tell whether the list is sorted unless it is run through at least one time. That is why the **REPEAT** is used here, instead of the **WHILE** loop.

7. Improve the sorting routine code.



Note: To this point the sorting routine is sufficient. It works well for small arrays. However, it can take any size arrays. If there are 100 elements, 1,000 elements, or 10,000 elements, a more efficient sorting routine is required.

Nested loops are created when one loop (the **FOR** loop) is put inside another loop (the **REPEAT** loop). With nested loops, there is always a potential for inefficiency, especially as the number of elements increase. The question is how many times the innermost loop executes. The LoopCount variable keeps track of this.

In the previous sorting routine, when the **FOR** loop is executed once, the **IF** statement is executed nine times, because the length of the array is 10 and it goes to two. If the **REPEAT** loop must execute nine times, the **FOR** loop executes nine times and the **IF** statement executes $9 * 9 = 81$ times. However, if the array length goes to 100, the **IF** statement executes 99 times for each **FOR** statement, and the **FOR** statement may be executed by the **REPEAT** loop up to 99 times. In this situation, the **IF** statement could be executed up to $99 * 99 = 9,801$ times.

You can reduce the execution significantly by remembering what happened when the **FOR** loop executes. Every time that it runs, the lowest element rises to the top. That means that the first element never has to be tested again after the first **FOR** loop runs. After the second **FOR** loop runs, you do not have to test the first two elements again. Each loop needs fewer and fewer tests to do its job. This is the worst case scenario.

- a. In the **Action Designer**, go to the next line and type the following to add another action. Make sure that the action is indented under the **ActionItems ActionContainer**.

| Caption | Type |
|---------------|--------|
| Improved Sort | Action |

- b. Click **View > C/AL Code** to open the **C/AL Editor**.
- c. Locate the **OnAction** trigger for the **Improved Sort** action.
- d. Type the following code into the **OnAction** trigger of the action:

Code Example

```

LoopCount := 0;
SwapCount := 0;
LowestSwitch := 2;

REPEAT
  IsSorted := TRUE;
  FOR idx := ARRAYLEN(OutputNumber) DOWNTO LowestSwitch DO BEGIN
    LoopCount += 1;
    IF OutputNumber[idx] < OutputNumber[idx-1] THEN BEGIN
      TempNumber := OutputNumber[idx];
      OutputNumber[idx] := OutputNumber[idx-1];
      OutputNumber[idx-1] := TempNumber;
      SwapCount += 1;
      IsSorted := FALSE;
      LowestSwitch := idx + 1;
    END;
  END;
UNTIL IsSorted

```

- e. Close the **C/AL Editor**, and then close the **Action Designer**.
- f. Compile, save, and close the page.



Note: Instead of sorting down to two (comparing with one) each time, now the **FOR** loops sorts down to LowestSwitch (comparing with LowestSwitch - one). LowestSwitch starts out as two. But every time that two values are switched (in the **THEN** clause of the **IF** statement), it resets to one more than the element that is currently being switched.

Because the **FOR** loop works down at the end, LowestSwitch is the lowest element to test the next time. Even in the worst case scenario, it is one greater than the previous value, and it might be several greater.

For example, if the array has ten elements in the worst case scenario, it executes the innermost **IF** statement $9 + 8 + 7 + 6 + 5 + 4 + 3 + 2 + 1$ times = 45 times. If the array has 100 elements, the worst case goes from 9,801 executions down to 4,950.

In other words, this simple modification makes the sort runs twice as fast, and therefore takes half as long to execute.

8. Test the page.
 - a. Run page **90007, Test Arrays Page**.
 - b. Type values into the input column, and then click **Populate Output**. View the results.
 - c. Click **Sort**, and then view the results.

- d. Click **Populate Output**, and then click **Improved Sort**. View the results.
- e. Click **Clear**, and then view the results.
- f. Click **Generate Input**, then click **Populate Output**, and then click **Sort**. View the results.
- g. Click **Populate Output**, and then click **Improved Sort**. View the results.

The WITH Statement

Use the **WITH** statement to facilitate coding with record variables. Understanding the concepts behind record variables lays the groundwork for understanding the **WITH** statement.

Record Variables

A *record variable* is a complex data type. Similar to an array variable, a record variable contains multiple values. In an array, all the values have the same name and type and are distinguished by an element number, or index. In a record, each value, known as a field, has its own name and type. To distinguish these values, a period (.) is used to separate the name of the record variable from the name of the field. Therefore, if a record called **Customer** has a field that is named **Name**, use **Customer.Name** to access that field.

A record represents a row or a record in a database table. The fields that are part of a record are defined by using a Table object.

The Syntax of the WITH Statement

The following code sample shows how to assign data in an array to a record variable:

Code Example

```
Customer.Name := Txt[1];  
  
Customer.Address := Txt[2];  
  
Customer.City := Txt[3];  
  
Customer.Contact := Txt[4];  
  
Customer."Phone No." := Txt[5];
```

Use the **WITH** statement to make this kind of code easier to write and easier to read. It has the following syntax:

Code Example

```
WITH <record variable> DO <statement>
```

Within the statement, which can be a compound statement, the record name is no longer required. Instead, the fields of that record can be used as if they were variables. Therefore, you can rewrite the previous code sample by using the **WITH** statement as follows:

Code Example

```
WITH Customer DO BEGIN  
  
    Name := Txt[1];  
  
    Address := Txt[2];  
  
    City := Txt[3];  
  
    Contact := Txt[4];  
  
    "Phone No." := Txt[5];  
  
END;
```

The Microsoft Dynamics NAV Development Environment associates the fields together with the **Customer** record. If there are variable names that are the same as the field names, the development environment uses the field from the record, instead of the variable. This ambiguous reference must be avoided.

Implied WITH Statements

In many locations in code you cannot see the **WITH** statement. These locations use the implied **WITH** statements. You can find examples of implied **WITH** in various objects.

For example, in a table object there is an implied **WITH** statement, that covers the whole object. Every reference to a field that is defined in that table has the implied record variable, known as **Rec**, and a period automatically and invisibly added in front of it.

Module Review

Module Review and Takeaways

There are many kinds of statements in C/AL. These include the following examples:

- Conditional statements, such as the **IF** and the **CASE** statement
- Compound statements that use the **BEGIN** and **END** construct
- Different kinds of repetitive statements, such as the **FOR** and the **REPEAT...UNTIL** statement

These statements are used everywhere in Microsoft Dynamics NAV. They provide the foundation for writing a complete application.

In addition to simple variables, arrays and records introduce a different way to store values. Both arrays and records are complex variables that hold multiple values. The use of the **WITH** statement eases the use of record variables to make code easier to read.

Understanding different kinds of statements helps developers decide the best method to write code to achieve certain functionalities.

Test Your Knowledge

Test your knowledge with the following questions.

1. This is a valid use of arrays: You have a list of students, numbered from 1 to 99. Each student takes a test. You want to put the numeric score (from 0 to 100) for each student into an array, with student 1's score going in element 1, student 2's score in element 2, and so on.

 True

 False
2. This is a valid use of arrays: You have a list of students, numbered from 1 to 99. Each student takes a test. You want to create a two-dimensional array with two rows of 99 columns. In the first row, put the corresponding student's name, with student 1's name in element 1,1, student 2's name in element 1,2, and so on. In the second row, put the corresponding student's numeric test score (from 0 to 100), with student 1's score going in element 2,1, student 2's score going in element 2,2, and so on.

 True

 False

3. This is a valid use of arrays: You create an array containing the number of households in each ZIP code. There is one array element for each five-digit ZIP code and each element contains the number of households in that ZIP code. The number of households that have a ZIP code of 30071 goes into element 30071, and so on.

True

False

4. This is a valid use of arrays: You create an array containing the number of households in each state. There is one array element for each two-character postal state code and each element contains the number of households in that state. The number of households in Georgia goes into element 'GA', and so on.

True

False

5. Which repetitive statement enables you to repeat one or more statements a known number of times?

6. Which repetitive statement does not require a BEGIN and END to execute more than one statement repetitively?

7. Which repetitive statement tests the condition at the beginning of each loop?

8. Rewrite the following WHILE statement as a REPEAT statement. Describe the differences in how it is executed.

```
WHILE X > 0 DO BEGIN  
  
    Y := A * X * X + B * X + C;  
  
    X := X - 1;  
  
END;
```

Test Your Knowledge Solutions

Module Review and Takeaways

1. This is a valid use of arrays: You have a list of students, numbered from 1 to 99. Each student takes a test. You want to put the numeric score (from 0 to 100) for each student into an array, with student 1's score going in element 1, student 2's score in element 2, and so on.

True

False

2. This is a valid use of arrays: You have a list of students, numbered from 1 to 99. Each student takes a test. You want to create a two-dimensional array with two rows of 99 columns. In the first row, put the corresponding student's name, with student 1's name in element 1,1, student 2's name in element 1,2, and so on. In the second row, put the corresponding student's numeric test score (from 0 to 100), with student 1's score going in element 2,1, student 2's score going in element 2,2, and so on.

True

False

3. This is a valid use of arrays: You create an array containing the number of households in each ZIP code. There is one array element for each five-digit ZIP code and each element contains the number of households in that ZIP code. The number of households that have a ZIP code of 30071 goes into element 30071, and so on.

True

False

4. This is a valid use of arrays: You create an array containing the number of households in each state. There is one array element for each two-character postal state code and each element contains the number of households in that state. The number of households in Georgia goes into element 'GA', and so on.

True

False

5. Which repetitive statement enables you to repeat one or more statements a known number of times?

MODEL ANSWER:

FOR

6. Which repetitive statement does not require a BEGIN and END to execute more than one statement repetitively?

MODEL ANSWER:

REPEAT...UNTIL

7. Which repetitive statement tests the condition at the beginning of each loop?

MODEL ANSWER:

WHILE...DO

8. Rewrite the following WHILE statement as a REPEAT statement. Describe the differences in how it is executed.

```
WHILE X > 0 DO BEGIN  
  
  Y := A * X * X + B * X + C;  
  
  X := X - 1;  
  
END;
```

MODEL ANSWER:

```
REPEAT  
  
  Y := A * X * X + B * X + C;  
  
  X := X - 1;  
  
UNTIL NOT (X > 0);
```

The WHILE statement executes only if the condition is TRUE at the beginning of the execution. The REPEAT statement executes at least once before checking the condition.