Chapter 1
Introduction to C/AL Programming

This is the introduction to Navision's Client Application Language.

This chapter will cover the following:

1.1 The C/AL Programming Unit

This unit will walk you through Navision's programming language and our development environment. You will become familiar with creating variables, using the Pascal-based language, and Navision's built in functions to run simple code. This will provide a good base in preparation for attending the Solution Developer class. The following topics will be presented in this course:

This course is part of the self-study program. Thoroughly read the material and take as much time as you need to complete each exercise. If you need to go back and review a chapter, take time to do that. This material will be tested.

1.2 What is C/AL?

Definition

C/AL (**C**lient **A**pplication **L**anguage) is the programming language used in the Client / Server Integrated Development Environment (C/SIDE) included with Navision Solutions.

What is it used for?

There are many purposes for which you can use a computer programming language. However, many of these uses are already handled for you by using the standard C/SIDE objects. For example:

- Data presentation is handled through the form objects and report objects.
- Data acquisition is mainly handled through form and dataport objects.
- Data storage and organization is handled by the table objects in conjunction with the built-in Database Management System (DBMS).

In C/SIDE, the main purpose of the programming language is data manipulation. Through C/AL, you can create business rules to insure that the data stored in the tables are meaningful and consistent with the way your customer does business. You can add new data or transfer data from one table to another (for example, a journal to a ledger). If data from multiple tables need to be combined onto one report or displayed on one form, you will probably need to program this.

Another purpose of C/AL is to control the execution of the various C/SIDE objects. With C/AL you are able to coordinate them in a way that meets the business needs of your customer.

Where is it Used?

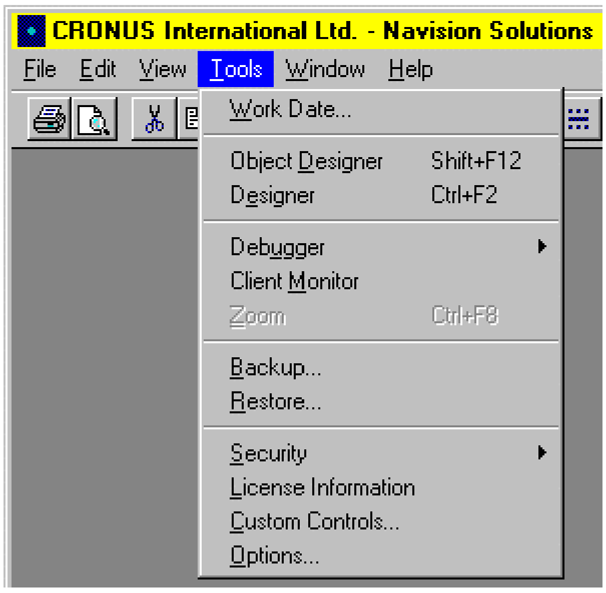
C/AL programming can be found in any Navision Solutions application object. In fact the codeunit application object is used only for programming. If you go into the Object Designer, press the codeunit button, select a codeunit and press the Design button, you will immediately see the C/AL editor and programming language statements (also known as “C/AL statements”, or just “code”).

Other objects can have C/AL statements as well, although they don’t always have them. This code is found in “triggers” within the object. Please start Navision Solutions and follow along as we explore.

1.3 Accessing C/AL

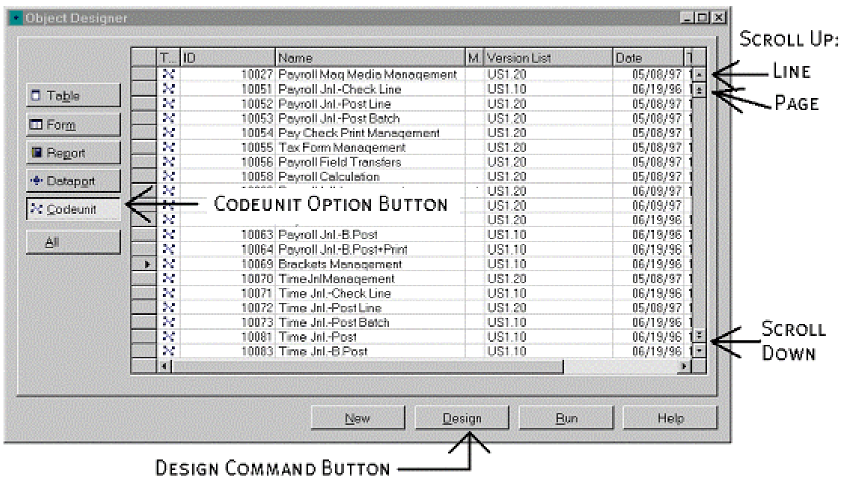
Start the Object Designer

To start the Object Designer, select the **Tools** option from the Menu Bar and then select **Object Designer** from the list that drops down.



View Codeunit Objects

To view codeunit Objects, press the **Codeunit** Option Button on the Object Designer form.



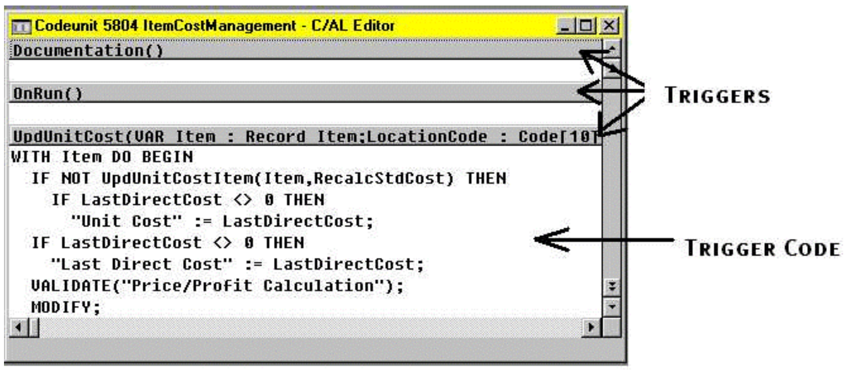
T..	ID	Name	M.	Version List	Date
	10027	Payroll Mag Media Management		US1.20	05/08/97
	10051	Payroll Jnl.-Check Line		US1.10	06/19/96
	10052	Payroll Jnl.-Post Line		US1.20	05/08/97
	10053	Payroll Jnl.-Post Batch		US1.20	05/08/97
	10054	Pay Check Print Management		US1.20	05/08/97
	10055	Tax Form Management		US1.20	05/08/97
	10056	Payroll Field Transfers		US1.20	05/08/97
	10058	Payroll Calculation		US1.20	05/08/97
	[illegible]	[illegible]		US1.20	06/09/97
	[illegible]	[illegible]		US1.20	06/09/97
	[illegible]	[illegible]		US1.20	06/19/96
	10063	Payroll Jnl.-B.Post		US1.10	06/19/96
	10064	Payroll Jnl.-B.Post+Print		US1.10	06/19/96
	10069	Brackets Management		US1.10	06/19/96
	10070	TimeJnlManagement		US1.20	05/08/97
	10071	Time Jnl.-Check Line		US1.10	06/19/96
	10072	Time Jnl.-Post Line		US1.20	05/08/97
	10073	Time Jnl.-Post Batch		US1.10	06/19/96
	10081	Time Jnl.-Post		US1.10	06/19/96
	10083	Time Jnl.-B.Post		US1.10	06/19/96

Select an Object

Using the picture above as a guide, scroll through the codeunits to find codeunit number 5804, ItemCostManagement, then click on it. Or, if you prefer, use the up and down arrows on your keyboard to move to codeunit 5804.

Look at the C/AL Code

Using the below picture as a guide, press the **Design** Button once codeunit 5804 - ItemCostManagement has been selected. The C/AL Editor window will appear, looking something like this:



```
Documentation()

OnRun()

UpdUnitCost(VAR Item : Record Item;LocationCode : Code[10]
WITH Item DO BEGIN
  IF NOT UpdUnitCostItem(Item,RecalcStdCost) THEN
    IF LastDirectCost <> 0 THEN
      "Unit Cost" := LastDirectCost;
  IF LastDirectCost <> 0 THEN
    "Last Direct Cost" := LastDirectCost;
  VALIDATE("Price/Profit Calculation");
  MODIFY;
```

Each gray bar that you see is called a "trigger". The C/AL code that you may see listed below the gray bar is the "trigger code" for that trigger. If there is no C/AL code between one trigger and the next trigger, then that trigger (in this case, OnRun) is said to be empty. For example, there is no trigger code between the OnRun trigger and the UpdUnitCost trigger.

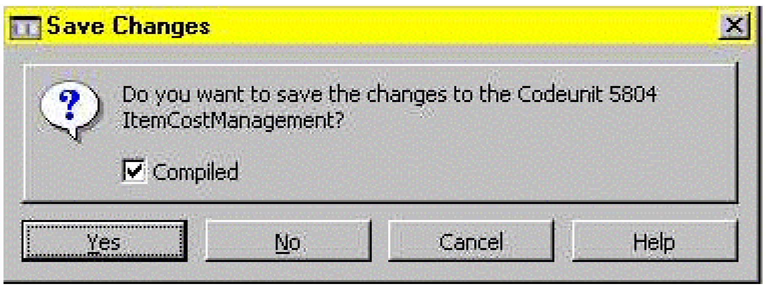
There are three kinds of triggers that you see here. The first is the "Documentation Trigger". This is not really a trigger and no code in this trigger will be run. Instead, you can use the Documentation trigger to write any sort of documentation you want for this object. Many people use this space to document their modifications to standard objects. Every object has a Documentation trigger.

The second kind of trigger is an “Event Trigger”. The name of these triggers always starts with “On”. The C/AL code in an event trigger is executed when the named event occurs. For example, the code in the OnRun event trigger is executed whenever this codeunit object is run. In this case, since there is no trigger code, nothing would happen. Each object has its own set of predefined event triggers that can be programmed.

The third kind of trigger shown here is a “Function Trigger”. These triggers are created whenever you create a “function” in an object. The C/AL code in this function trigger is executed whenever the function is “called”. You will learn more about creating and calling functions in another section. You will learn more about event triggers in your Solution Developer class.

Closing an Object

After you have looked at the ItemCostManagement codeunit, close the C/AL Editor window by clicking the Close Box or by pressing the ESC key. If you have not changed anything, the object will be closed and you can continue. If you have changed something, the following box will pop up:



Save Changes

Do you want to save the changes to the Codeunit 5804 ItemCostManagement?

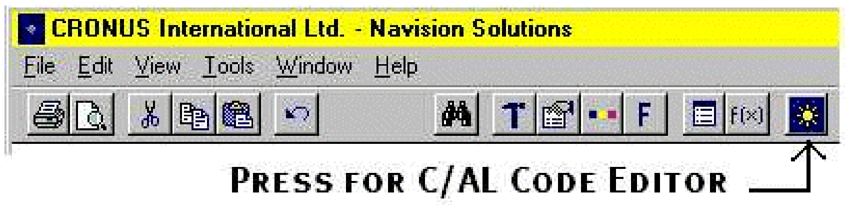
☑ Compiled

For now, press the **No** button (or press the ALT+N key) to discard any changes you may have made in order to exit the C/AL Editor and close the object.

Look at C/AL Code in a Table Object

Press the **Table** Button in the Object Designer window. Scroll through the Object Designer window and select Table 18 (the Item table). Press the **Design** button to open the Table Designer window. Note that this is not the C/AL Editor. Instead, it is a list of the fields that make up the table definition. You will learn more about these later.

To get to the C/AL Code, you must open the C/AL Editor. To do this, review the Tool Bar:


CRONUS International Ltd. - Navision Solutions
File Edit View Tools Window Help
PRESS FOR C/AL CODE EDITOR

Press the C/AL button. As an alternative, you can press the F9 key. Once you are in the C/AL Editor, note that there are numerous triggers in this Table Object. Most of them are empty, but several of them contain C/AL code.

Most of these are in triggers labeled with a field name followed by "OnValidate". These are event triggers and the code is triggered by the "event" of the user completing the entry of that field. You will learn more about this and the other triggers found here in your Solution Developer class. For now, just take a look around and then close the C/AL Editor window. Once it is closed, you can close the Table Designer, again without saving any changes you may have accidentally made.

1.4 Self Test

Terminology

1 What is the programming language of C/SIDE called?

2 List three or more uses of programming code:

3 Where can programming language statements be found?

4 What do you use to modify code in an object?

5 List the three basic types of triggers:

6 What is the key you press to view or modify code in an object (other than a codeunit object)?

Practical

Use the Object Designer to look at the code in Form Object 39.

Use the Object Designer to look at the code in Report Object 321. Note that the code will be different, depending upon which line (also called a “DataItem”) is selected in the Report Designer. See what happens when you select the line following the last line in the Report Designer before starting the code window.

Self-Test Answers

Terminology

1 What is the programming language of C/SIDE called?

C/AL (**C**lient **A**pplication **L**anguage)

2 List three or more uses of programming code.

Data Manipulation
Data Presentation
Data Acquisition
To control the execution of C/AL objects
To implement business rules, etc.

3 Where can programming language statements be found?

In Application Objects (e.g. Forms, Tables, Reports, Codeunits and Dataports). Also, in the triggers within those Application Objects.

4 What do you use to modify code in an object?

The C/AL Editor

5 List the three basic types of triggers.

Documentation Triggers, Event Triggers ,Function Triggers.

6 What is the key you press to modify code in an object (other than a codeunit object)?

F9

Practical

Form Objects – Note that the Form itself has triggers, and also that the individual controls in the Form also have triggers.

Report Objects – Each DataItem has different triggers. The Report triggers are found when you select the blank line following the last DataItem in the Report Designer.

Chapter 2
Simple Data Types

This section will introduce the concept of data types, explain what the different kinds of data types are, when to use them and how to assign them.

This chapter will cover the following:

2.1 Simple Data Types

Definitions

Data

Data is known facts, known pieces of information. For our purposes, we will always use “data” to mean information that is available for us to manipulate in Navision Financials using C/SIDE.

Data Types

Data Types are the different kinds of information that may appear in C/SIDE. Different Data Types have different values, different meanings for those values and are manipulated differently. For example, if we have two different data values - “25” and “37” - and we add them, we will get different results depending on what type of data they are. If they are numbers, the result would be “62”. If, on the other hand, they are text, the result could be “2537”.

Constants

Constants are data values that are written directly into our programming statements. They are called Constants because their values never change while the user is running the application. Constants can only be changed by changing the C/AL code.

Simple Data Types

Simple Data Types are those types of data which have only one value and which cannot be broken up into other values of different types.

Byte

A Byte is a unit of data storage space used in computers. One character stored in the computer takes up one byte of storage. Related terms are a Kilobyte (KB), which is 1024 bytes, a Megabyte (MB), which is 1024 Kb or 1,048,576 bytes and a Gigabyte (GB), which is 1024 Mb, or 1,073,741,824 bytes.

2.2 Numeric Data Types

Numeric data types are all forms of numbers or amounts. As such, there are many automatic methods used by C/SIDE to convert one type of number to another behind the scenes and in many cases they can be used interchangeably. However, in some cases, their differences can be quite important, sometimes causing errors and sometimes causing more subtle problems.

Integer

An integer is a whole number that can range in value from -2,147,483,647 to +2,147,483,647. It takes up 4 bytes of storage. By default, an integer is equal to zero. Typical constants of type Integer in C/AL are:

- 12
- 1000 (note that there are no commas as they are invalid in numeric constants)
- -100
- 0

Decimal

A decimal is a whole or fractional number that can range in value from $-1x10^{63}$ (a 1 followed by 63 zeroes) to $+1x10^{63}$, with values as small as $1x10^{-63}$ (a zero followed by a decimal point followed by 62 zeroes followed by a 1). It is kept with up to 18 digits of precision in Binary Coded Decimal (BCD) format and takes up 12 bytes of storage. Also, the default value of a decimal is zero. Typical constants of type Decimal in C/AL are:

- 12.50
- 52000000000 (again, note that there are no commas)
- -2.0
- 0.008
- -127.9533

Option

An option is a special kind of Integer that allows the programmer to define words for each value. For example, if we created a variable called Spectrum, we could assign it as an option with the following OptionString:

- Red,Orange,Yellow,Green,Blue,Indigo,Violet

The default value of an option is zero, since it is an integer and this represents the 1st element, which is “Red”. Therefore, “Green” is represented by the integer 3. Note that there are not any spaces between the elements, as a space would become part of the element’s name.

Char

A char is a single character. It takes up 1 byte of storage. For syntax purposes, it is considered a numerical type and can have integer values from 0 to 255. It can be used along with other numerical types in expressions. Typical constants of type Char in C/AL are:

- 'b' (note the single quotes surrounding the character)
- 'C'
- '3'
- '?'

2.3 String Data Types

String data is data that is made up of strings of characters. The data that is placed in word processors is string data. In spreadsheets, where most of the data is considered numeric, string data is sometimes entered using a special prefix to distinguish it. In C/AL constants, the symbol used to distinguish string data is the single quote, also known as an apostrophe ('). All string constants are surrounded by single quotes.

Text

A text is a string of 0 to 250 characters. The **length** of a text is the number of characters in it. To find the amount of storage a text takes up, add one to the length and round up to the nearest four. Thus an 8-character text takes up 12 bytes (1 + 8 rounded up to the nearest 4). Typical constants of type Text in C/AL are:

- 'Hello'
- '127.50' (even though this looks like a number, since it is surrounded by quotes, it is a text)
- '' (Note that this is an empty, 0 length text)
- ' spaces before ... and after '
- 'Here''s how to use an apostrophe' (note that to put a single apostrophe in a text constant, insert two apostrophes)

Code

A code is a special kind of text. All letters are forced to upper case and all leading and trailing spaces are removed. In addition, when displayed to the user, a code is automatically right justified if all characters in it are numbers. To find the amount of storage a code takes up, add two to the length and round up to the nearest 4. Thus an 11-character code takes up 16 bytes (2 + 11 rounded up to the nearest 4). The same text constants above, converted to code, look like this:

- 'HELLO'
- '127.50'
- ''
- 'SPACES BEFORE ... AND AFTER'
- 'HERE''S HOW TO USE AN APOSTROPHE'

When comparisons are done on two code values, or when they are sorted, the special right justification feature mentioned above is taken into account. Thus, for codes, '10' is greater than '9', but '10A' is less than '9A'. For texts, '10' is less than '9' as well.

2.4 Boolean, Date and Time

Boolean

Boolean data, also known as logical data, is actually the simplest form of data. It takes up only 1 byte in memory, though when it is stored in the database it uses 4 bytes. The constants of type Boolean in C/AL are only

- TRUE
- FALSE

Note that if these values are compared, the False value is less than the True value because it is stored as a zero and True is stored as one. However, the integer value is not interchangeable with the constant of TRUE or FALSE. In code, the Boolean variable must be set to "TRUE" or "FALSE", not zero or one.

Date

A date is just what it says, a calendar date, which can range in value from 1/1/0000 through 12/31/9999. It takes up 4 bytes of storage. In addition, the value of a date can either be a **Normal Date** or a **Closing Date**. The Closing Date represents the last millisecond of the last minute of the last hour of the day, so it is greater than the Normal Date with the same calendar value. Typical constants of type Date in C/AL are:

- 123197D (December 31, 1997)
- 030595D
- 08171953D (August 17, 1953)
- 0D (The undefined date, less than all other dates)
- 063012D (June 30, 2012)

All these Date constants are Normal Dates. There are no Closing Date constants in C/AL.

The general syntax is mmddyyD or mmddyyyyD. Two digits may be used for the year, which will be translated differently depending upon which version of Navision you are using. For versions 2.6 and greater, if the year is from 30 to 99, it is considered to be in the 1900's and if it is from 00 to 29, it is considered to be in the 2000's. For version 2.01 through 2.5, the year 19 is considered

2019, but year 20 is 1920.

Note that the date is defined with a 'D' at the end. If there is no 'D', then C/AL assumes that it is an integer and an error will occur as one can't assign an integer to a date. Also, in code, do not use slashes to separate the month, day and year. That implies division and you will get an error since it can't assign an integer or decimal to the date variable.

Time

A time data type represents the time of day (not a time interval) which can range in value from 00:00:00 through 23:59:59.999. It takes up 4 bytes. Typical constants of type Time in C/AL are:

· 103000T (10:30am)

· 154530T (3:45:30pm)

· 0T (The undefined time, less than all other times)

· 030005.100T (3:00:05.1am)

· 225930.135T (10:59:30.135pm)

The general syntax is hhmmss[.xxx]T, where the fractions of seconds (.xxx) are optional. Similarly to the date type, the time type must have a 'T' at the end of it to distinguish it from an integer.

2.5 Simple Data Types - Self-Test

1) What data type should be used to store an employee's birthday?

2) What data type should be used for an employee’s name?

3) What data type should be used for an employee’s weekly salary (you need to record it to the penny)?

4) What data type should be used to record whether or not an employee is income tax exempt?

5) Write down the data type of this constant: 'You must enter a positive value.'

6) Write down the data type of this constant: 123197

7) Write down the data type of this constant: 327.01

8) Write down the data type of this constant: 3,498

Simple Data Types - Answers and Review

1) What data type should be used to store an employee's birthday?

   Date

2) What data type should be used for an employee’s name?

   Text

3) What data type should be used for an employee’s weekly salary (you need to record it to the penny)?

   Decimal

4) What data type should be used to record whether or not an employee is income tax exempt?

   Boolean

5) Write down the data type of this constant: 'You must enter a positive value.'

   Text

6) Write down the data type of this constant: 123197

   Integer

7) Write down the data type of this constant: 327.01

   Decimal

8) Write down the data type of this constant: 3,498

   This is an invalid constant, due to the comma appearing outside of quotes.

Chapter 3
Identifiers and Variables

This section reviews identifiers and variables. You will see the syntax of identifiers, how to initialize variables and understand their scope. This also defines what system defined variables are.

This chapter will cover the following:

3.1 Identifiers and Variables

Definitions

Identifier

An identifier is a name for something, a way to identify it. Objects, variables, fields and functions all have identifiers. These Identifiers must be used when referring to these entities. Not all items found in a program have Identifiers. Among those that do not are constants, operators and certain reserved words. Instead, these items are referred to directly. One way to understand the difference is that those programming elements that refer to something stored elsewhere in memory require an identifier to access them, while those that exist in the programming code itself and do not refer to anything outside, do not need an Identifier.

Variable

A variable is the reference to a data value that can vary while the user is running the application. A variable refers to an actual location in memory in which data is stored. A variable has a name, also called the identifier, which the programmer uses in the program rather than an actual memory address. A variable also has a data type, which describes the kind of data that can be stored in that memory address. Finally, a variable has a value, which is the actual data currently stored in that memory address.

Syntax

Syntax is the set of grammatical rules that define the programming language. Programming lines that follow these rules are said to follow the proper syntax. The computer does not understand programming lines that do not follow the proper syntax; they cannot be compiled or executed. We have already defined the proper syntax for constants in section 2. More syntactical rules will be covered in this unit.

3.2 The Syntax of Identifiers

There are two ways that valid identifiers can be constructed in the program. The first is to follow the proper Pascal syntax. This means that the first character in the identifier must be either an underscore (_) or a letter (either upper or lower case). Following this first character, you can have up to 29 additional characters, each of which must either be a letter (upper or lower case), an underscore, or a digit (a number from 0 through 9).

The second method is used if you do not want to follow the normal Pascal syntax. In this case, the identifier must be surrounded by quotation marks (") when used within C/AL. Then, you can use any characters except "control" characters (characters whose underlying ASCII code is from 0 - 31 or 255) and the quote character itself ("). You can even use spaces.

Note that if you use the second method, the number of characters in an identifier can still include 30 characters and that these 30 do not include the quotation marks. Secondly, C/SIDE does not distinguish between upper and lower case letters in identifiers. Thus if there were two identifiers, the first being "Account Number" and the second being "Account number", these two would be seen as identical by C/SIDE.

Which brings up another point. Within one object, all identifiers must be unique. An identifier cannot be the same as one of the reserved words (like BEGIN or END) or an operator (like DIV or MOD). If so, a syntax error will result. In addition, two identifiers should not have the same name in an object, unless there is some other way to distinguish them. If a reference is "ambiguous" (that is, if C/SIDE cannot tell what exact programming element you are referring to), it will result in an error.

3.3 Variable scope and initialization

Global and Local Variables

All variables have a defined "scope", that is, a defined set of places where it can be accessed. A variable is said to have "global" scope if it can be accessed anywhere in an object. A variable is said to have "local" scope if it can only be accessed in a single trigger in an object. No variables can be accessed outside of the object in which they are defined.

System Defined Variables

Certain variables are automatically defined and maintained by the system. An example of this is the variable called Rec in table objects. Each object has its own set of system-defined variables. The programmer can use these variables whenever they need to without doing anything special to either create them or to initialize their values.

Variable Initialization

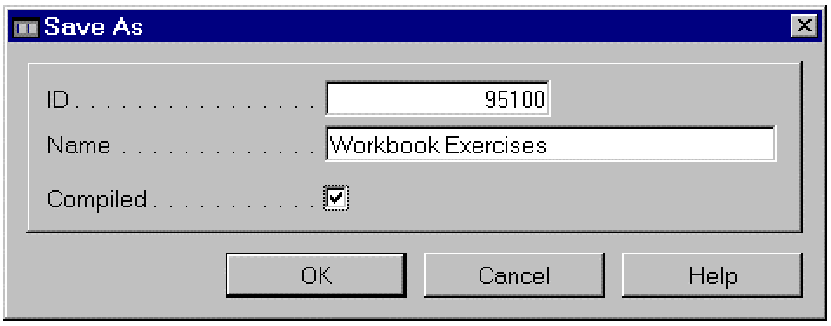
Before any C/AL code is executed, all programmer-defined variables are initialized with certain values.

- For all variables with a numeric type, this value is 0.
- For string variables, the value is the empty string ('').
- For Boolean variables, the initial value is FALSE.
- For Date and Time type variables, this initial value is 0D (the undefined date) and 0T (the undefined time) respectively.

3.4 Create a New Codeunit

Exercise

Open the Object Designer by selecting **Tools, Object Designer** from the Menu Bar. Press the **Codeunit** button to view all codeunits. Now press the **New** button located at the bottom of the Object Designer. The C/AL Editor window will appear, completely blank. Now close the C/AL Editor window. A question window will appear asking "Do you want to save the changes to the codeunit"? Click **Yes**. The Save As window will now be displayed. Fill it in as follows:

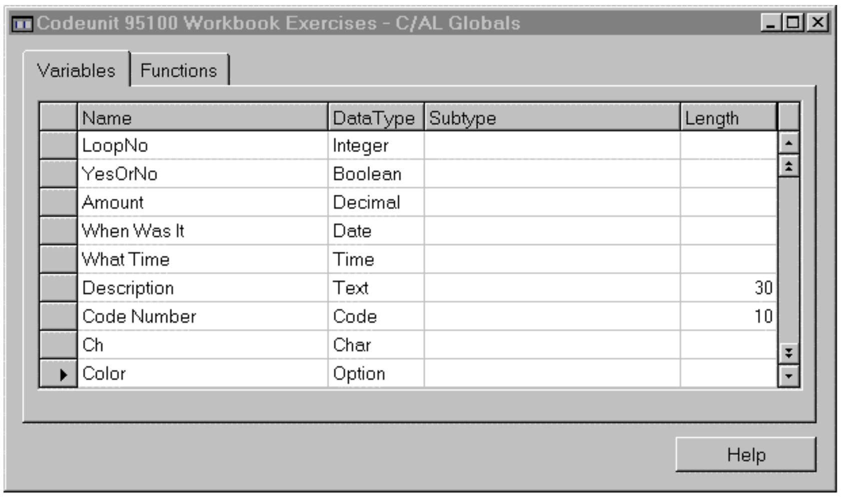


ID 95100
Name . . . . Workbook Exercises
Compiled . . . . ✓

Click **OK**. Now, scroll down to the bottom of the Object Designer window and find your new codeunit. Select it and press the **Design** button.

Define a Variable

To define global variables in an object, select **View, C/AL Globals** on the Menu Bar once you are designing that object. A window like the following will appear:

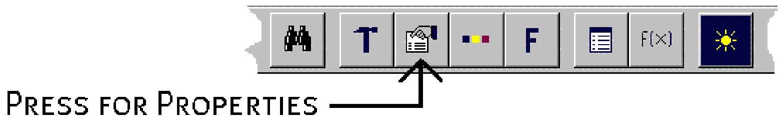

Codeunit 95100 Workbook Exercises - C/AL Globals

Name	DataType	Subtype	Length
LoopNo	Integer		
YesOrNo	Boolean		
Amount	Decimal		
When Was It	Date		
What Time	Time		
Description	Text		30
Code Number	Code		10
Ch	Char		
Color	Option		

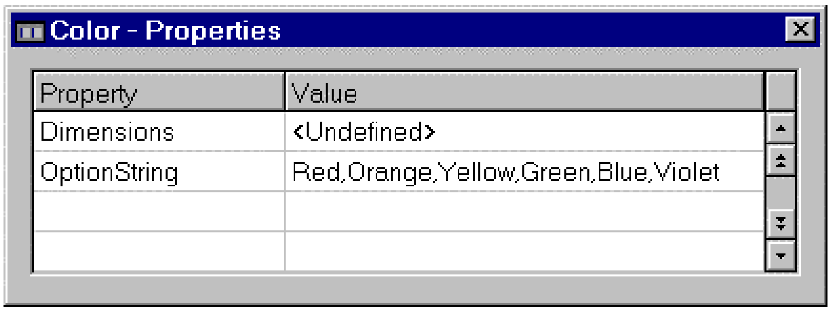
Please fill it in as shown above. If you press the F6 key while you are in the **DataType** column, you will see a list of data types that you can select. Note that these simple data types make up only a portion of the types that can be created.

Once you have entered all of these, select the Variable named Color. Note that in the picture above, Color is already selected, as shown by the black triangle in the left margin.

Now click the Properties button on the Tool Bar:



A window like the following should appear. Fill in the OptionString property exactly as shown in this window:


Color - Properties

Property	Value
Dimensions	<Undefined>
OptionString	Red,Orange,Yellow,Green,Blue,Violet

Note that commas separate the words, but that there are no spaces between the commas and the words. Now close the Properties window, then close the C/AL Globals window. Finally, close the Object Designer window. When it asks if you want to save, click **Yes**.

3.5 Displaying Variables

Click **Design** and go back into our Workbook Exercises codeunit. Click on the first line underneath the OnRun Trigger. Now type in the following line of C/AL code:

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

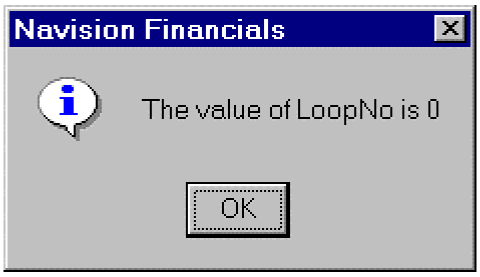
The MESSAGE function allows you to display a simple window with a single message in it, plus an **OK** button for users to press when they have finished reading the message.

Note that following the function identifier MESSAGE, you'll find an open parenthesis, a text constant, a comma, another text constant, another comma, the LoopNo variable, a close parenthesis and a semicolon. When a function like this is **Called**, it is followed by its **Parameters** enclosed in parentheses. Here there are three parameters, each one separated from the others by a comma.

The first parameter is text and contains the actual message to be displayed. This text has special substitution strings in it. A substitution string is allowed in certain functions, including this one, and consists of a percent sign followed by a single digit from 1 to 9. When this function is actually executed at run time, the first substitution string (%1) is replaced by the first parameter following it, and the second substitution string (%2) is replaced by the second parameter following it. If the parameters are not text parameters (for example, the LoopNo parameter above), it is automatically converted or formatted into text first before substitution is done.

Close and save the codeunit and then click **Run**.

The following window will appear:



The value of LoopNo is 0

Note that the first substitution string was replaced by the second parameter (the first parameter following the message), the text constant 'LoopNo'. The second substitution string was replaced by the third parameter (the second parameter following the message), which was the value of the variable LoopNo.

Note that the value of the integer variable was automatically initialized to zero and that this is the value that was substituted into the message. Click **OK** to clear the message window.

3.6 Identifiers and Variables - Self-Test

1 Modify the code line in the Workbook Exercises codeunit to display the name of your second variable (YesOrNo) and its value. Run the codeunit and write down the resulting message here.

2 Do the same for Amount.

3 Do the same for When Was It. Don't forget to enclose the variable name in quotes when you use it in the third parameter.

4 Do the same for What Time.

5 Do the same for Description.

6 Do the same for Code Number.

7 Do the same for Ch.

8 Do the same for Color.

Lesson 3: Identifiers and Variables - Answers and Review

1 Modify the code line in the Workbook Exercises codeunit to display the name of your second variable (YesOrNo) and its value. Run the codeunit and write down the resulting message here.

The value of YesOrNo is No

Note that when converted to text TRUE becomes 'Yes' and FALSE becomes 'No'.

2 Do the same for Amount.

The value of Amount is 0

3 Do the same for When Was It. Don't forget to enclose the variable name in quotes when you use it in the third parameter.

The value of When Was It is

Note that since the date is undefined, when it is converted to a text, the text is empty. The code line should read:

```
MESSAGE('The value of %1 is %2','When Was
It',"When Was It");
```

4 Do the same for What Time.

The value of What Time is

5 Do the same for Description.

The value of Description is

6 Do the same for Code Number.

The value of Code Number is

7 Do the same for Ch.

The value of Ch is

Note that when a Ch is converted to text, its zero value is converted to an empty text.

8 Do the same for Color.

The value of Color is Red

Note that the first word written in the OptionString property corresponds to the zero value for the option.

Chapter 4
The Assignment Statement

This chapter explains how to assign values to a variable using the correct syntax. You will see how to convert data from one type to another and see the syntax of statement separators. The Symbol Menu is introduced, which is used to assist with code writing.

This chapter contains the following sections:

4.1 Assignment Statements

Most developers are familiar with the assignment and statements terms, but, again, for clarification, we will define them here.

Assignment

Assignment means setting a variable to a value. Once a variable has a value, it keeps that value until it is set to another value or until the current set of code ends and then the computer no longer keeps track of the variable. C/AL has several assignment methods, one of which is the simple assignment statement.

Statement

A programming statement is a single complete programming instruction written into code. You can think of it as a code line, since normally we write one statement per line. In reality, however, one statement could take up many actual lines in the C/AL Editor and, similarly, many statements could be on one line in the C/AL Editor.

Assignment Statement

The assignment statement is a particular type of statement. This statement specifically assigns a value to a variable.

4.2 Assigning a Value to a Variable

Being able to assign variables to different values is a cornerstone for programming. Different programming languages may have different syntax for doing this. The below section describes C/AL's method.

Assignment Syntax

We have already seen that certain function calls, like the MESSAGE function introduced in Lesson 3, can be a statement all by themselves. We call these "Function Call Statements". The syntax for a Function Call Statement is very simple:

```
<function call>
```

The result of calling a function will vary depending upon which function is called. Likewise, the syntax of the function call itself varies with the function being called.

The syntax of an assignment statement is almost as simple as the function call statement:

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

The "colon equals" (:=) is called the assignment operator. The assignment statement evaluates the expression and the variable is set to this resulting value. We will cover expressions more fully in future lessons, but for now, we will use either a constant or another variable on the right side of the assignment operator. A constant or a variable can be used as a very simple expression.

4.3 Automatic Type Conversions

In order for a variable to be assigned a value, the type of the value must match the type of the variable. However, certain types, within limits, can be converted automatically during the assignment operation.

String Data Types

Both the string data types, code and text, can be automatically converted from one to the other. For example, if Description was a variable of type text and "Code Number" was a variable of type code, the following statement would be valid:

```
"Code Number" := Description
```

The text value in Description would be converted into code before being assigned to the "Code Number" variable. This means that all the lower case letters would be converted to upper case and all leading and trailing spaces would be deleted. Thus, the value assigned to the code variable would be of type code. Note that this conversion process does not affect the value stored in Description. Variables on the right side of the assignment operator are not modified by the assignment operation, only the one variable on the left side of the assignment operator is modified.

Note

There are limits to what can be done with the automatic conversion. For example, suppose that the value of the Description text variable had more characters than could fit in the "Code Number" code variable. In this case, an error would result when this statement was executed while the program was running, also known as a **Run-Time Error**.

Numeric Data Types

All of the numeric types (integer, decimal, option and char) can be converted automatically from one to another, as well. There are also restrictions here:

- A decimal value must be a whole number (no decimal places) in order to be converted to any of the other numeric types.

- Among the other types, if the value falls outside of the range of the variable type, the conversion will not take place. For example, a decimal cannot be converted to an integer or an option unless the value is between negative 2,147,483,647 and positive 2,147,483,647, which is the valid range for integers.

- A decimal, integer or option cannot be converted to a char unless the value is between 0 and 255, since that is the valid range for chars. If any of these conversions are attempted, a run-time error will result.

Other Data Types

Although string types can be automatically converted from one to the other and the same with numeric types, no other variable types can be converted automatically from one to another using assignment.

4.4 The Statement Separator

As stated previously, a single programming statement might cover several code lines and that one code line might have multiple statements. If so, how does the C/AL compiler tell when one statement ends and another statement begins? It recognizes a statement separator as the indicator of a statement ending. The statement separator is the semicolon (;). When the compiler reaches a semicolon (or the end of a trigger) it knows that it has reached the end of a statement. Here is the syntax of a trigger:

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

The brackets are used to indicate that whatever is enclosed in them is optional; it can either be there or not. Whatever is enclosed in the braces is deemed optional and can be repeated zero or more times. In other words, it is optional for any statements to appear in a trigger and as many statements as desired can appear in the trigger, as long as each one is separated from the others by a semicolon.

Note that whenever a semicolon is used, a statement must follow. But what about at the end of a trigger, where you often end the last statement with a semicolon? In this situation, a **Null Statement** is automatically inserted after the last semicolon and before the end of the trigger. As its name implies a Null Statement is nothing and does nothing. Although this seems like a moot point now, it is important to note that the semicolon does not signal the end of a statement (it is not a "statement terminator"), but instead signals the arrival of a new statement (it is a "statement separator"). This distinction will be critical in understanding the syntax of certain other statement types.

4.5 Using Assignment Statements and The Symbol Menu

The following exercise will give you practice using assignment statements and introduce you to the very useful symbol menu tool.

Simple Assignment Statement

Go into the Object Designer, select Codeunit 95100 and click the **Design** button. In the OnRun trigger, enter the following two statements, removing any code that was there previously.

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

Now exit, save and **Run** the codeunit.

Multiple Messages

Go back and edit codeunit 95100 again. After the code you entered above, add the following lines:

```
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");
```

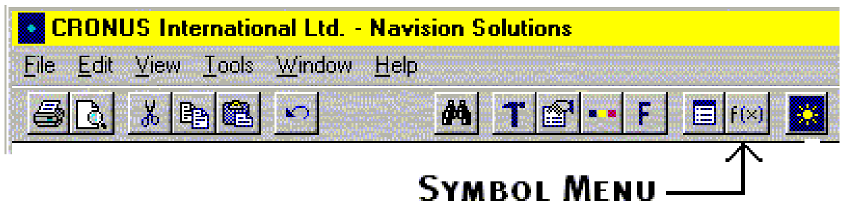
Exit, save and **Run** the codeunit.

Note that the messages will appear sequentially, one after the other. As you click **OK** for each message, the next message will display. Also, when you look at the last message (the one displaying the Code Number), note that automatic type conversion took place. The leading and trailing spaces were stripped and the letters turned to upper case when the value was converted from type text to type code.

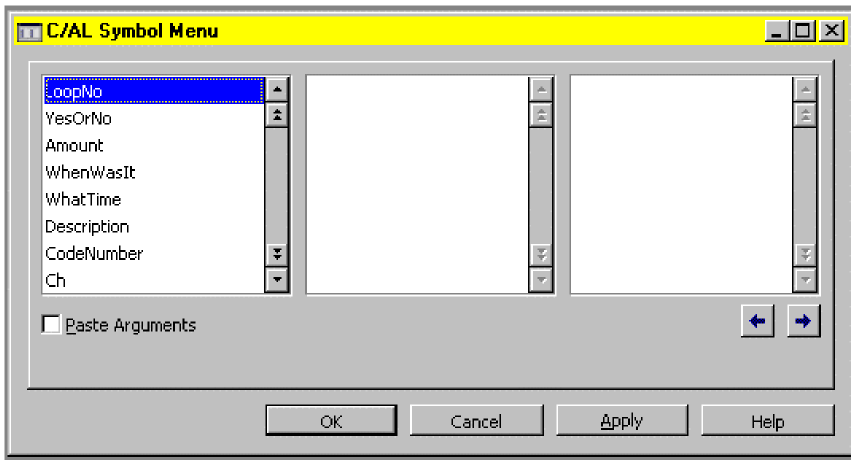
One thing that is not apparent, however, is that messages do not display until after the processing has ceased. In other words, as the MESSAGE functions are called, messages are added to a queue. A *queue* is like a line at a ticket counter: people get into the line at one end and leave the line at the other end in the same order they entered it. After the codeunit has completed its execution, the messages are read off the queue and displayed on the screen in the order that they were put into the queue. Because messages do not display immediately, they do not interrupt the processing. However, this also means that they cannot be used to give the user feedback on processing in progress. They can only be used to give the user results of processing.

Use of the Symbol Menu

We are now going to add a few more lines to codeunit 95100, using a slightly different method. Go to the end of the existing code, just as if you were about to type in another line. Instead, select **View, C/AL Symbol Menu** from the Menu Bar, press the F5 key, or press the Symbol button on the Tool Bar.


CRONUS International Ltd. - Navision Solutions
File Edit View Tools Window Help
SYMBOL MENU

The following window should appear:



In the left panel, you'll see a list all of the identifiers defined by the programmer, among other things. Click on the line which says "YesOrNo", then click OK. The Symbol Menu will disappear and the word `YesOrNo` will appear in your code.

After YesOrNo, type in:

```
:= TRUE;
```

so that the line now says:

```
YesOrNo := TRUE;
```

Then press ENTER and continue on the next line:

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

With the cursor still sitting just after the single quote you last typed, press the F5 key again, select YesOrNo again in the Symbol Menu (it should still be selected from last time) and click **OK**. Continue with a single quote and a comma. Again, press F5 and this time press the ENTER key (which also presses the **OK** button on the Symbol Menu). Finish off by typing a close parenthesis and a semicolon. The result should be:

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

Try using this method again for Description, but set the value (after the Assignment Operator) to 'Now is the time.' The result should be:

```
Description := 'Now is the time. ';
MESSAGE('The value of %1 is %2',
'Description',Description);
```

Again, use the Symbol Menu to enter another two lines setting and displaying "What Time". Set its value to 153000T. The result should look like this:

```
"What Time" := 153000T;
MESSAGE('The value of %1 is %2','"What Time"',"What
Time");
```

Note that by using the Symbol Menu, the double quotes are automatically inserted when needed. In this case, however, the double quotes inside the single quotes are not needed, so you should go back and remove them. The result will be:

```
"What Time" := 153000T;
MESSAGE('The value of %1 is %2','What Time',"What
Time");
```

Now, exit and save the codeunit and run it again.

Note that the Time and Date displays are dependent on the settings you made in your system Control Panel (Regional Settings). On most US systems, for example, the time would be displayed as 3:30:00 PM. However, a simple change on your Control Panel will change this display to 15:30:00. Note that this does not require a code change, or even a recompile, under C/SIDE. The Control Panel change itself is sufficient.

Char Constants

Because of automatic type conversion, a Char variable can be set using either a number or a one character text. Enter the following lines in your codeunit:

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

Note that both 65 and 'A' result in the same value displayed in the message. This is because 65 is the **ASCII** code for the upper case A. You may want to experiment with a few other numbers to see what the results are. These codes are used for characters in C/SIDE, as well as in most other programs and computer systems.

Option Constants

We previously covered using a number as an option constant. In C/SIDE, another syntax can be used to write option constants into your code. Enter the following lines into your codeunit:

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

Once you have saved and executed this new code, you will see that the results are the same for both messages. The first option is always numbered 0, so the third option, in this case Yellow, is numbered 2.

Note that the syntax used for option constants is the Variable Identifier, followed by two colons, followed by the Option Identifier.

Run-Time Errors

As we mentioned before, not every value can be converted automatically to a value of another type.

Try entering these lines, one at a time, into your codeunit, compile and run it. See what kinds of error messages you get:

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

Note that only the first one and the last one will actually compile, as the compiler is smart enough to figure out that some things would never work. In the first case, however, since both the constant and the variable were of type text, it wasn't until you executed the codeunit that it discovered the overflow (Description was defined as 30 characters long, but the constant is 36). In the last case, the fact that Amount contained a value that could not be converted to integer could also not be found until the program was executed. These kinds of errors are called **Run-Time Errors**. Note that when a run-time error is discovered, it stops all further processing and produces an error message.

4.6 Self-Test

Feel free to use your practice codeunit to help you answer these questions. All variables refer to those you defined in that codeunit.

1) What would be displayed if you displayed Amount after setting its value to 15.00?

2) What if you then assigned Amount to LoopNo and then displayed LoopNo?

3) What would be displayed if you set Color to the value of 4 and then displayed Color?

4) What if you set Color to 8 and then displayed it?

5) What would be displayed if you set Description to 'the time.', then assigned Description to Code Number and then displayed Code Number?

6) What would be displayed if you set Description to 'Now is the time.', then assigned it to Code Number and then displayed Code Number?

7) What would be displayed if you set Color to Green, assigned Color to LoopNo and then displayed LoopNo?

Self Test - Answers and Review

1) What would be displayed if you displayed Amount after setting its value to 15.00?

   15

2) What if you then assigned Amount to LoopNo and then displayed LoopNo?

   15

3) What would be displayed if you set Color to the value of 4 and then displayed Color?

   Blue

4) What if you set Color to 8 and then displayed it?

   8

5) What if you set Description to 'the time.', then assigned Description to Code Number and then displayed Code Number. What would be displayed?

   THE TIME.

6) What if you set Description to 'Now is the time.', then assigned it to Code Number and then displayed Code Number. What would be displayed?

   A run-time error (text overflow).

7) What would be displayed if you set Color to Green, assigned Color to LoopNo and then displayed LoopNo?

   3

Chapter 5
Expressions

This chapter defines expressions, operators, terms and evaluation. It explains how string operators work and how to use function calls in expressions. The operator precedence for numeric expressions and the effect of binary operators on data types are also reviewed.

This chapter contains the following sections:

5.1 Expressions, Terms and Operators

We will start with a review of the definitions of these terms, and in later sections, we will see these in use.

Expression

An **Expression** is like an equation put into a program, a formula telling the computer exactly how to generate the desired value. Like a variable and a constant, an expression has a type and a value. However, unlike either one, an expression must be evaluated at run-time in order to determine its value. Although a constant's value is known at all times, a variable's value is determined at run-time, the system must look it up in memory. The following are examples of expressions:

- FunctionX + 7
- Quantity * UnitCost

Evaluation

To **Evaluate** an expression means to actually follow the instructions set out in the formula and 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 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 surrounded by parentheses. This last type of term is also known as a subexpression. The above 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 (i.e. a unary operator), or the terms on either side of it (i.e. a binary operator). Operators are represented by symbols (e.g. +, >, /, =) or reserved words (e.g. DIV, MOD). They are defined by the C/AL language and cannot be redefined or added to by the programmer. During expression evaluation, when the operator operates on its term(s), it results in another value, which may be the value of the expression, or may be a term used by another Operator. Some examples of operators and their uses:

- 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

5.2 The Syntax of an Expression

Describing an expression's syntax requires a few more syntax statements than we have seen previously, since it is so complex.

The ::= symbol used below means that the element on the left side of that symbol has a syntax defined by what is on the right side of that symbol. Note that the vertical bar (|) means "or". Any one of the elements separated by vertical bars will work as 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> }

Remember that the braces surrounding an element mean zero or more repetitions of that element. We will be covering the various operator types as we cover different expression types in this and future lessons.

5.3 The String Operator

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

Expression Example

Here is an example of an expression and how it is evaluated. First a couple of assignments, then the expression:

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

The expression is evaluated as follows:

First, the value of `CodeA` is obtained.

Then, the constant value is concatenated to the end. Since `CodeA` is a Code while the constant is a Text (note the trailing spaces), 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. Then, this value is concatenated to the end of the previously generated text. Since both values are Text, the type of the result is Text and the value is `'HELLO THERE. How Are You? '` . Since this is the end of the expression, this result becomes the result of the expression, both the type and the value.

Once the expression is evaluated, the new value is assigned to the `CodeB` variable using the assignment operator. Since the type of the expression is Text and the type of the `CodeB` is Code, the result of the expression must be converted to Code using automatic type conversion. The result of this conversion is that `'HELLO THERE. HOW ARE YOU?'` is assigned to CodeB.

String Operator Exercise

Go into the Object Designer, select codeunit 95100 and click **Design**. Add the following global variables to the current variable list, by selecting **View, C/AL Globals** on the Menu Bar:

```
CodeA Code  30
CodeB Code  50
TextA Text  50
```

```
TextB Text  80
```

In the OnRun trigger, enter the following four statements, after removing any code that was in that trigger from previous lessons.

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

Now exit, save and **Run** the codeunit. You should get the same result as was described above.

5.4 Function Calls in Expressions

Remember that if you try to assign too many characters to a string variable, you will get a run-time error. How can this be prevented, since the error does not occur until the program is running? One way is to design your program such that this error could never happen. This is how it is handled in most of Financials. In some cases, though, this is impossible. Fortunately, there is another method.

MAXSTRLEN Function

There is a function available that will tell you (at run-time) the maximum length of a string that can fit in a variable. This function is called `MAXSTRLEN`. It has one parameter, which is the variable in question. The return value is of type integer.

Enter the following two code lines into codeunit 95100, after the lines entered above:

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

The result will be that LoopNo will be set to 30, which is the length that you used when you created this variable.

Now that we know how long a string we can put into this variable, how do we put only that many characters into it? First let's look at the result if we do not do this. Type in the following lines after the ones above:

```
Description := 'The message is: ' + CodeB;
MESSAGE('The value of %1 is
%2','Description',Description);
```

Once you save and run this, you will get the expected run-time error. The problem is that the assignment statement transfers the entire value of the expression on its right to the variable on its left and there is not enough space in this case. What we need is something that will only get part of a text.

That something is a function called `COPYSTR`.

This function has three parameters.

- The first is the string that is being copied from.
- The second is the position of the first character to copy. If we wanted to start copying from the fifth character, this second parameter would be 5. If we want to start from the first character, the second parameter would be 1.

- The third parameter is the number of characters to copy. This third parameter is optional and if it is not there, then all the characters of the string are copied from whatever the second parameter says to the last character. If you try to copy more characters than are in the string, there is no harm done. The maximum number of characters that will be copied is the number in the original string. The value of the COPYSTR function, once it has been evaluated, is a copy of the first parameter, but only including those characters designated by the second and third parameters. If the first parameter is a Text, the type of the COPYSTR function is Text. If the first parameter is a Code, the type of the COPYSTR function is Code.

Now, in your codeunit, find the line you typed that looks like this:

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

Change it to look like this:

```
Description := COPYSTR('The message is: ' +
CodeB,1,MAXSTRLEN(Description));
```

When you save and run this, you will no longer get a run-time error. Instead, the value of Description will only include the characters that would fit.

How Did it Work?

Let us evaluate this expression "by hand" and see how it worked.

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

```
COPYSTR('The message is: ' +
CodeB,1,MAXSTRLEN(Description))
```

First, to evaluate a function like COPYSTR, we must evaluate each of its parameters. The first step, to evaluate the first parameter, is to get the value of CodeB and concatenate it (note the plus sign) with the string constant. The result is this:

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

The next step is to evaluate the second parameter. Since this is a constant, it is easy. Next, we must evaluate the third parameter. This parameter is a function that returns the defined length of its parameter (which must be a variable, not an expression). The result is:

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

Finally, the COPYSTR function itself can be evaluated. In this case, it 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:

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

Now that the expression has been evaluated, the assignment can be performed:

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

Description now has its new value.

5.5 Self-Test

1) Underline 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) Underline the expression in this assignment statement:
```
TextA := 'The ' + TextB;
```

5) Underline the operator(s) in the expression:
```
TextA := 'The ' + TextB;
```

6) Underline the term(s) in the expression:
```
TextA := 'The ' + TextB;
```

Self Test - Answers

1) Underline the expression in this assignment statement:

   TextA := $\underline{\text{TextB}}$;

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

   Terms

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

   It concatenates them.

4) Underline the expression in this assignment statement:

   TextA := $\underline{\text{'The ' + TextB}}$;

5) Underline the operator(s) in the expression:

   TextA := 'The ' $\underline{+}$ TextB;

6) Underline the term(s) in the expression:

   TextA := $\underline{\text{'The '}}$ + $\underline{\text{TextB}}$;

Chapter 6
Numeric Expressions

This chapter reviews the arithmetic operators available in C/AL and explains the operator precedence for numeric expressions. This chapter contains the following sections:

6.1 Numeric Expressions and Operator Precedence

The terms “Numeric Expressions” and “Operator Precedence“ are used frequently in this chapter. Here are the definitions.

Arithmetic Operators

Arithmetic operators are those operators that are used in numeric expressions to operate on numeric or non-numeric terms. Addition, subtraction, multiplication, and division symbols are arithmetic operators.

Numeric Expression

This type of expression, which uses at least one arithmetic operator, results in a numeric data type. So, when evaluating a numeric expression, the result is of type decimal, integer, option or char. Although the individual terms of a numeric expression might not be numeric, the result will be.

Operator Precedence

This is the order that operators are evaluated within the expression. Operators with a higher precedence are evaluated before operators with a lower precedence. For example, since the multiplication operator (*) has a higher precedence than the addition operator (+), the expression 5 + 2 * 3 evaluates to 11, rather than 21, which is what it would evaluate to under the normal left to right rule.

6.2 Types of Arithmetic Operators

There are six different arithmetic operators in C/AL.

The Plus (+) Operator

The plus operator is used for several purposes. As shown earlier, if the terms on either side of it are both string types, the result is a string, so this would not be an example of it being used as an arithmetic operator. However, it can be used as either a unary or binary arithmetic operator.

When used as a unary operator, its function is to not change the sign of the term following it. It literally does nothing and is rarely used. If it is used, its purpose is to explicitly show that a value is positive. Here is an example of an expression using the plus unary operator and the same expression without it:

- IntVariable * +11
- IntVariable * 11

When normally used as a binary operator, its function is to add the term following it to the term preceding it. Both terms can be numeric, or one term can be a Date or a Time and the other an integer. If either term is a decimal value, then 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 allowable values for integers, the result is integer; otherwise, the result is decimal.

If one term is a date and the other an integer, the result is the date that is the integer number of days away from the date term. Thus, the value of `03202001D + 7` is `03272001D`. If the resulting value results in an invalid date, a run-time error will occur.

Similarly, if one term is a time and the other an integer, the result is the time that is the integer number of milliseconds away from the time term. Thus, the value of `115815T + 350000` is `120405T`. If the resulting value results in an invalid time, a run-time error will occur.

Technically the plus operator used with a date or a time term is not an example of an arithmetic operator, since the result is not numeric. However, we have covered it here for convenience. The chart below reviews this information.

Note that the left column indicates the type of the term preceding the plus operator, while the top row indicates the type of the term following the plus operator.

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
Time	Time	Time	Time	N/A	N/A	N/A

Remember that if a result would normally be a char but the value is not a valid char value, the result type changes to integer. If a result would normally be an integer but the value is not a valid integer value, the result type changes to decimal.

The Minus Operator (-)

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

When used as a binary operator, its function is to subtract the term following it from the term preceding it. Both terms can be numeric, both can be a date or a time, or the preceding term can be a Date or a Time, while the following term is an integer.

If the first term is a date and the second is an integer, the result is the date that is the integer number of days before the date term. Thus, the value of `02252001D - 7` is `02182001D`. If the resulting value results in an invalid date, a run-time error will occur.

If the first term is a time and the second is an integer, the result is the time that is the integer number of milliseconds before the time term. Thus, the value of `115815T - 350000` is `115225T`. If the resulting value results in an invalid time, a run-time error will occur.

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 chart summarizes the result types when the minus operator is used on various term types. Note that the left column indicates the type of the term preceding the minus operator, while the top row indicates the type of the term following the minus operator:

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
Date	Date	Date	Date	N/A	Integer	N/A
Time	Time	Time	Time	N/A	N/A	Integer

Remember that if a result would normally be a char but the value is not a valid char value, the result type changes to integer. If a result would normally be an integer but the value is not a valid integer value, the result type changes to decimal.

The Times Operator (*)

The times operator (or the multiplication operator) is only used as a binary operator. Its function is to multiply the numeric term preceding it by the numeric term following it. The following chart summarizes the result types when the times operator is used on various term types:

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 normal automatic conversion rules, from char to integer to decimal, apply.

The Divide Operator (/)

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

The Integer Divide Operator (DIV)

The integer divide operator is also only used as a binary operator. Its function is to divide the numeric term preceding it by the numeric term following it. The result type of this division is always integer. If the second term is zero (0), the result is a run-time error. Any decimals that would have resulted from an ordinary division are dropped, not rounded. Thus, the result of `17 DIV 8` is `2`, while the result of `17 DIV 9` is `1`.

The Modulus Operator (MOD)

The modulus operator requires two numbers. The first number is the one that is converted using the modulus function and the second number. The second number represents the number system being used. By definition, the number system starts at zero and ends at the second number minus 1. For example, if the second number were 10, then the number system used would be from 0 to 9. So, the modulus represents what the first number would convert to if your numbering system only had the number of values indicated by the second number, and then was forced to restart at zero. Here are some examples:

- 15 modulus 10 is 5 (since 9 is the last number available, 10 is represented by going back to the start, or zero, 11 is 1, 12 is 2, etc.).
- 6 Modulus 10 is 6
- 10 Modulus 10 is 0
- 127 Modulus 10 is 7

Notice that you would get the same result if you divided the first number by the second using integers only and returned the remainder as the value.

The modulus operator (or the remainder operator) is only used as a binary operator. Its function is to divide the numeric term preceding it by the numeric term following it using the integer division method outlined above and then return the remainder of that division. The result type of this operation is always an integer. If the second term is zero (0), the result is a run-time error.

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

Arithmetic Operator Exercise

Go into the Object Designer, select codeunit 95100 and click **Design**.

Add the following global variables to the current variable list, by selecting **View, C/AL Globals** on the Menu Bar:

```
Int1  Integer
Int2  Integer
Amt1  Decimal
Amt2  Decimal
```

In the OnRun trigger, enter the following statements, after removing any code that was in that trigger from previous lessons.

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

Now exit, save and **Run** the codeunit. The results you should get are that the LoopNo is 25 and Amount is 1.

6.3 Operator Precedence Effects

It is important to know the precedence that operators take, since unexpected results could occur if they are ignored.

Operator Precedence Levels

There are three levels of operator precedence used for arithmetic operators.

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

Normal evaluations of expressions proceed from left to right. However, if one operator has a higher precedence level than another, it is evaluated first. To override either of these methods, the programmer can create subexpressions by surrounding parts of an expression with parentheses. Subexpressions are always evaluated first.

The purpose of this exercise is to illustrate the various division functions, including normal division, integer division and modulus. Please examine the code and the results thoroughly, so that you understand how it works. If you need to, insert other messages to view the intermediate results.

Operator Precedence Exercise

Return to codeunit 95100 and add the following code lines after the lines you entered above:

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

Save it and run and you will find that the result is 24.

Using the precedence rules it was evaluated as follows:

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

```
Int1 := 5 + 18 - 2 DIV –2
```

2 The integer divide operator (DIV) is evaluated, dividing its preceding term (2) by its following term (-2) and resulting in a new term of –1:

```
Int1 := 5 + 18 – (-1)
```

3 The plus operator (+) is evaluated, adding its following term (18) to its preceding term (5) and resulting in a new term of 23:

```
Int1 := 23 – (-1)
```

4 Finally, the binary minus operator (-) is evaluated, subtracting its following term (-1) from its preceding term (23) and resulting in the value of the complete expression, which is 24 (23 minus a negative 1 is 24).

Now, create subexpressions by adding parentheses as follows:

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

Save it and run again and you will find that the result is negative 16. Knowing that subexpressions are evaluated first, it was evaluated as follows:

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

```
Int1 := 8 * (6 - 2) DIV -2;
```

2 The next subexpression is evaluated. The binary minus operator (-) subtracts its following term (2) from its preceding term (6), resulting in the value of 4 for the subexpression. Again, this value is now a term of the complete expression.

```
Int1 := 8 * 4 DIV -2;
```

3 The unary minus operator (-) and its following term (2) is evaluated, resulting in a value of negative 2 (-2).

```
Int1 := 8 * 4 DIV -2;
```

4 The times operator (*) is evaluated, multiplying its preceding term (8) by its following term (4) and resulting in a new term of 32.

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

5 Finally, the integer divide operator (DIV) is evaluated, dividing its preceding term (32) by its following term (-2) and resulting in the value of

the complete expression, which is negative 16 (-16).

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

6.4 Self-Test

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

1 57 * 10

2 57 / 10

3 57 DIV 10

4 57 MOD 10

5 2000000 * 3000

6 9 / 4 - 9 DIV 4

7 (3 - 10) * - 5 - 10 + 2.5 * 4

8 02201996D + 14

9 02101996D – 14

10 01201996D - 02101996D

Self Test Answers

1	57 * 10	Integer	570
2	57 / 10	Decimal	5.7
3	57 DIV 10	Integer	5
4	57 MOD 10	Integer	7
5	2000000 * 3000	Decimal	6000000000
6	9 / 4 - 9 DIV 4	Decimal	0.25
7	(3 - 10) * - 5 - 10 + 2.5 * 4	Decimal	35
8	02201996D + 14	Date	03051996D
9	02101996D - 14	Date	01271996D
10	01201996D - 02101996D	Integer	-21

Chapter 7
Logical and Relational Expressions

This chapter defines the logical and relational expressions and operators available in C/AL and their operator precedence. This chapter contains the following sections:

7.1 Logical and Relational Operators and Expressions

Logical and relational expressions are different from arithmetic expressions in that they always result in a Boolean value (either true or false). These expressions use logical or relational operators to determine their value. However, there are some differences between them, so it is important to fully understand these concepts.

Relational Operator

This operator is used in a relational expression to test a relationship between the term preceding it and the term following it, resulting in a Boolean value. The relational operators are:

- = (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, by using a relational operator, compares values and results in a Boolean value (True or False). The terms of this expression are usually not Boolean themselves, but they must be compatible with each other. Thus an integer can be compared with a decimal, since they are both numeric, but an integer cannot be compared to a text, since one is numeric and the other is string. Below are some examples:

- 5 <= IntVar
- DecVar <> IntVar

Logical Operator

This operator uses one or two Boolean terms in a logical expression. The logical binary operators are AND, OR and XOR (exclusive or). The one logical unary operator is NOT.

Logical Expression

This expression uses at least one logical operator, which results in a Boolean value. It is similar to a relational expression in that it can have terms of any type, but relational expressions must have two terms and one operator. A logical expression can have multiple terms, but the terms must all be of type Boolean. The following are examples of logical expressions:

- TRUE AND FALSE (results in a value of False)
- FALSE OR NOT FALSE (results in a value of True)
- (Quantity> 5) OR (Quantity <= 10) OR (Price < 100) (This has three terms)

7.2 Relational Expressions for Comparison

All of the relational operators, except for IN, compare two values. These two values must be of the same type, or of compatible types. All of the various numeric types (e.g. integer and decimal) are compatible. Both of the string types (text and code) are compatible.

Numeric Comparisons

If a comparison is done between two numbers, the normal numeric rules apply. Note the following examples:

· 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

String comparisons use a modified alphabetical order, not the ASCII order that many other programming languages use. One major difference you will note is that special characters used in other languages (like the é in Danish or the ñ in Spanish) are placed in their proper alphabetical order, not relegated to the end as they would in ASCII order. Another major difference is that the digits appear after the letters in alphabetical order, while in ASCII order, they come before the letters. Finally, in alphabetical order, the lower case letters come before the upper case letters, while in ASCII order, the lower case letters come after the upper case letters. Here are some examples:

· 'X' = 'X'	is	TRUE
· 'X' = 'x'	is	FALSE
· 'ark' > 'arc'	is	TRUE
· 'arC' > 'arc'	is	TRUE

·	'10' > '2'	is	FALSE
·	'00' <= 'OO'	is	FALSE
·	'é' > 'f'	is	FALSE
·	'abc' < 'ab'	is	FALSE
·	' a' <> 'a'	is	TRUE

Also, remember that when a code variable is used in a comparison, there are some special rules. All trailing and leading spaces are removed and all letters are converted to upper case. In addition, code values that consist only of digits are right justified before comparison. Thus, the fifth example above ('10' > '2') would evaluate to TRUE if those values were loaded into variables of type code before comparison.

Date and Time Comparisons

Date and Time values are compared as you would expect; dates (or times) farther in the future are greater than dates (or times) in the past. A Closing Date (which represents the last second of the last minute of that date) is greater than the Normal Date for the same day and less then the Normal Date for the next day.

Boolean Comparisons

Boolean values are normally not compared using relational operators, but they can be. When they are, True is considered greater than False.

Relational Operator Precedence

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

Thus, the following example evaluates to True:

5 * 7 < 6 * 6

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

7.3 Relational Expressions for Set Inclusion

The relational operator IN is used to determine inclusion. It determines if the first term is in a specific set. Therefore, it requires a list of values, a set, to compare the term to. This list is part of the expression and is called 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 open square bracket ([) followed by a list of allowed values separated by commas, followed by a close square bracket (]). For example, a set of all the even numbers from one to ten would look like this:

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

Besides 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 would look like this:

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

In addition, an individual value or a value used as part of a range can actually be an expression. A list of numbers from 10 to 20, but not including the variable n (as long as n was from 10 to 20), would look like this:

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

The IN Operator

Once you have constructed a set constant, the IN operator's operation is quite simple. It checks to see if the value of the term preceding it is included in that set. For example:

· 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

7.4 Using Logical Expressions

As stated before, a logical operator operates on Boolean terms and results in a Boolean value. Often, these Boolean terms are the result of a relational expression.

Logical Operator Results

The following reviews the logical operator results:

· The unary Boolean operator, NOT, logically negates the term following it, changing 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 either (but not both) of the terms on either side of it are True, but if both are True or both are False, it results in False.

The following "truth tables" summarize these facts:

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

Logical Operator Precedence

The NOT operator has the same precedence as the other unary operators, positive (+) and negative (-), which is the highest precedence. This means that it is evaluated before any other operator in the same expression. The AND operator has the same precedence as the multiplicative operators, while XOR and OR have the same precedence as the additive operators. As stated before, the relational operators have the lowest precedence of all.

The following table summarizes the operator precedence of all of the operators covered so far:

Type of Operator	Operator	Comments
Subexpression or Terms	( ) [ ] . ::	Sub-expression in parentheses is evaluated first. Other operators are used in a Term.
Unary	+ - NOT	Highest Precedence within Expression
Multiplicative	* / DIV MOD AND	
Additive	+ - OR XOR	
Relational	< <= = >= > <> IN	Lowest Precedence within Expression
Range	..	Used in Set Constants

When dealing with logical expressions, it is especially important to note that the relational operators are lower in precedence than the logical operators. Thus, if you want to see if a variable N is between 10 and 20 exclusive, you cannot use the following expression:

```
N >= 10 AND N <= 20
```

If you used the above, the first operator to be evaluated would be AND and the terms preceding (10) and following (N) are integer, not Boolean, terms. This will cause an error. Instead, you need to use parentheses to force the relational operators to be evaluated first:

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

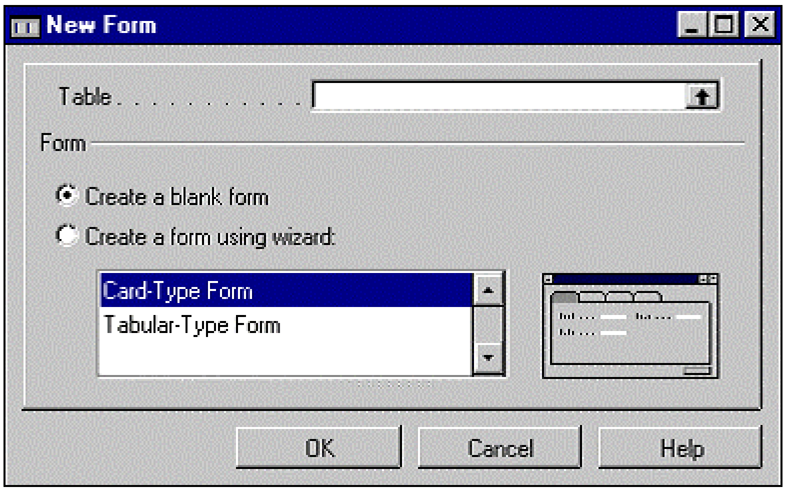
Now, the two relational expressions are evaluated first, resulting in two Boolean terms. Then the logical operator AND can be legally evaluated.

7.5 Adding Logical and Relational Expressions to a Form

In the past, we have worked with a simple codeunit to evaluate expressions and used the MESSAGE function to view the results. Now, in order to allow more data to be tested without as many recompiles, we will create a form object to work with.

Creating a New Form

Open the Object Designer by selecting **Tools, Object Designer** from the Menu Bar. Click the **Form** Button to view all forms. Now click the **New** Button located at the bottom of the Object Designer. The New Form window will appear, as shown below:



New Form

Table . . . .

Form

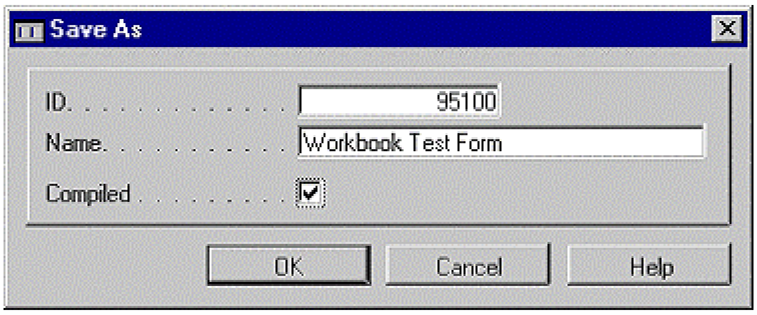
(●) Create a blank form

( ) Create a form using wizard:

Card-Type Form
Tabular-Type Form

Click **OK**. A blank form window will appear.

We will now save it so that it appears as a form object in the object designer window. Close the form window. A question window will appear asking "Do you want to save the changes to the Form"? Click **Yes**. The Save As... window will be displayed. Fill it in as follows:



Save As

ID. 95100

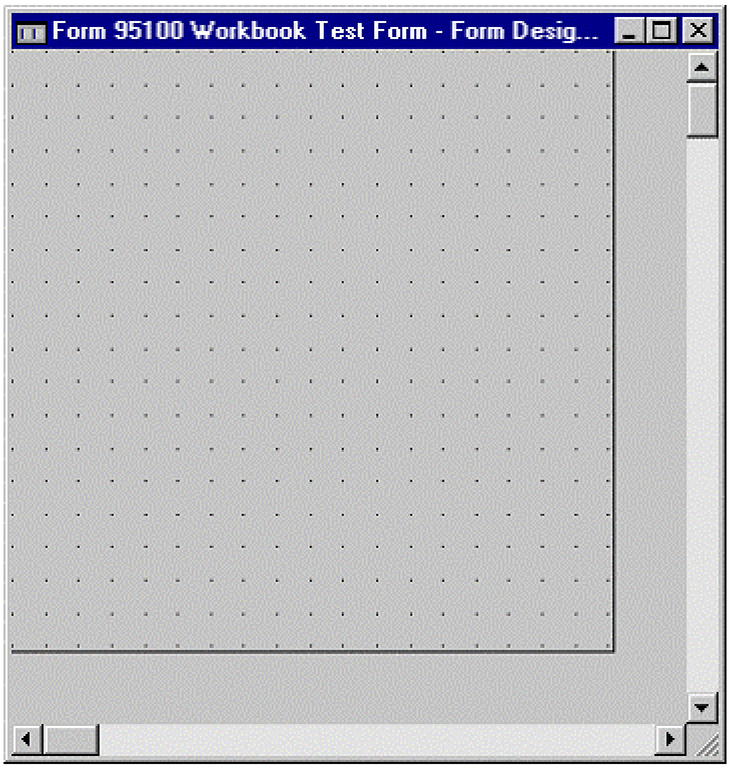
Name. . . . . Workbook Test Form

Compiled . . . . ✓

Then, click **OK**.

Scroll down to the bottom of the Object Designer window and find your new form.

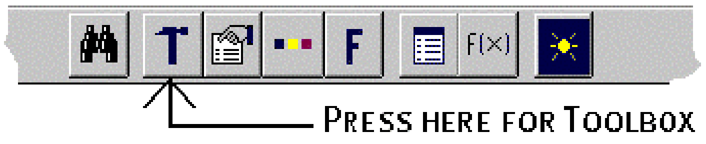
Select it and press the **Design** Command Button. The following blank form window will appear:



Adding Variables to a Form

Select **View, C/AL Globals** from the Menu Bar, and create two variables of type Integer called Value1 and Value2. Also, create a variable of type Boolean called Result. Close the C/AL Globals window.

Display the Toolbox window, by pressing the Toolbox button on the Tool Bar:



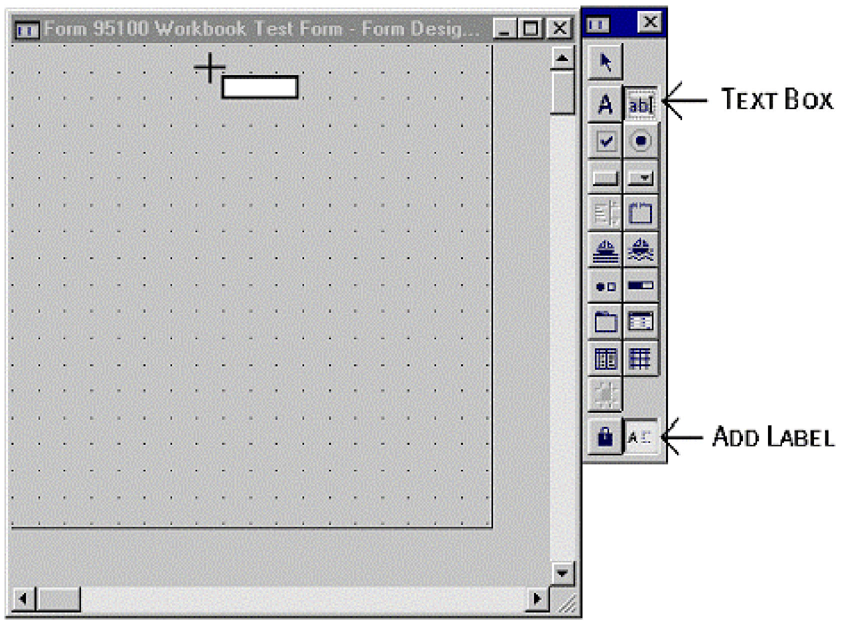
The Toolbox will appear.

Move it next to the blank form as shown below.

Select the Add Label tool in the lower right corner.

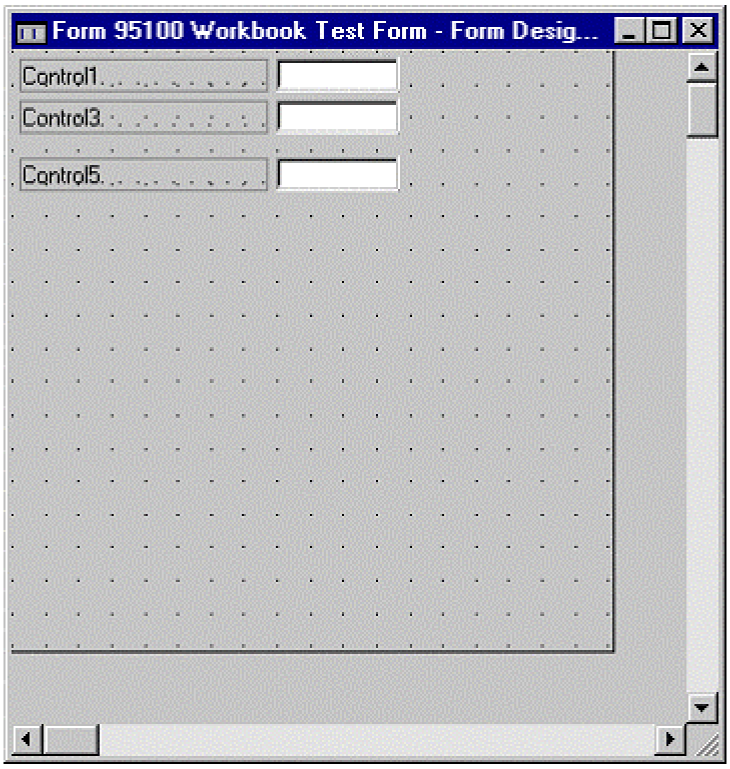
Select the Text Box tool in the right-hand column.

Move the mouse cursor until it is over the blank form, as shown in the following picture. The cursor should change into a special positioning cursor. The one you see will not look exactly like this picture, but it should be close. Position the cursor so that the cross is near the center top of the blank form. Click.



A text box "Control" will appear (the white box) and to the left of it will be the associated label "Control" (the gray box). Click on the Text Box tool again, move the positioning cursor to a spot just below the existing text box control, and click again. Finally, click on the Text Box tool again, move the positioning cursor little below the other two text boxes, and click.

You should end up with the form looking something like this:



If the controls did not line up neatly, you can move them by dragging the text box control into place. Note that when you drag the text box control, the associated label control (the gray box) moves with it.

Now, click on the first text box control and then press the Properties button on the Tool Bar.

The Properties button is the button just to the right of the Toolbox button. The Properties window will appear, as you have seen it before, but this time there will be more properties than you have seen before.

For now, we are only interested in two of them. Find the **Caption** property and change the value to "First Value". Find the **SourceExpr** property and change the value to "Value1". Then close the Properties window.

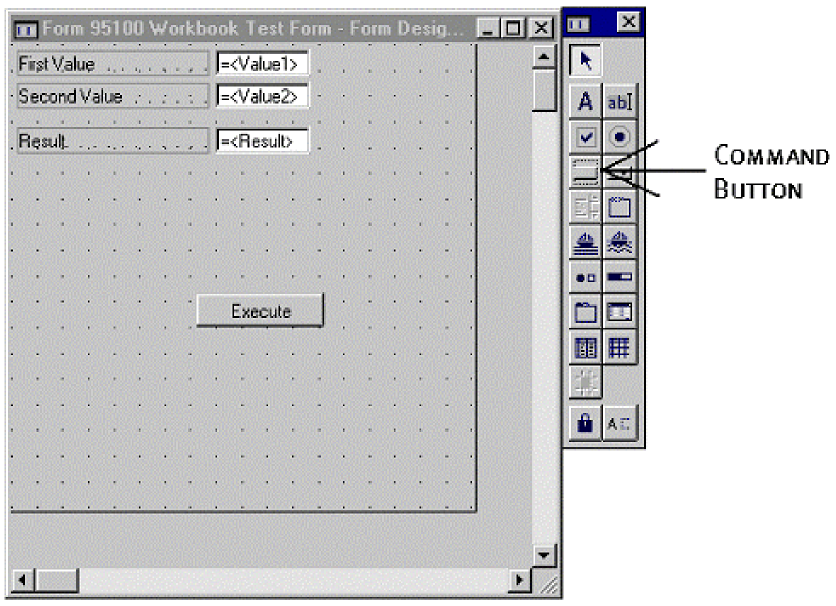
Click on the second text box control, then press the Properties button. Change this control's Caption property to "Second Value", and change the SourceExpr property to "Value2". Finally, click on the last text box control, display the Properties window, change the Caption property to "Result", and change the SourceExpr property to "Result".

Close the form, save it and run it.

Note that you can set any of the three values when running the form. The first two can be any integer value, while the third is limited to either Yes or No, since it is a Boolean. When a Boolean value is displayed as text, True displays "Yes", while False displays "No".

Adding a Command Button and C/AL Code

Open the form again by clicking the **Design** button. The form and the Toolbox should display. Click on the Add Label tool so that it is no longer selected. Now, click on the Command Button tool. Move the positioning cursor to the middle bottom of the form and click. Display the Properties window and change the Caption Property to "Execute". After closing the Properties window, the screen should look like this:



Click on the command button and press the C/AL Code button on the Tool Bar, or press the F9 key. Find the Trigger that says OnPush. This code will be executed whenever you push the command button. As an example of what you can do, type the following code into the OnPush trigger code section:

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

Now close the form, save and run it. Try entering a value into the First Value box, another one in the Second Value box and then press the Execute button. The Result box is automatically updated with the result of the expression you placed in the assignment statement in the OnPush trigger code. Try several values in each box, including some negative values.

Once you are comfortable that you can predict the results when you press the

Execute button, try putting this code into the OnPush trigger in place of the code above:

```
Result := (Value1 >= Value2) AND (Value1 <= 2 *
Value2);
```

Try other formulas as well. See what you can learn about how expressions are evaluated.

7.6 Self-Test

On all of the questions below, feel free to use the test form you created to find the answers. Remember that each control's SourceExpr property determines which variable is being displayed and updated through that control. For each question, write down the value of the result of evaluating the logical or relational expression:

1 5 * 7 > 35

2 5 * -7 > -36

3 (3 > 5 - 1) OR (7 < 5 * 2)

4 (27 MOD 5 = 2) AND (27 DIV 5 = 5)

5 (5 > 3) XOR (7 = 7) AND (9 >= 10)

6 (10 > 2) AND ('10' > '2')

7 NOT (11 + 7 < 15) OR ('Great' > 'Greater') AND ('Less' < 'Lesser')

8 TRUE OR FALSE = TRUE

9 Given that X is a Boolean Variable:

NOT (X AND TRUE OR FALSE) = (NOT X OR FALSE) AND TRUE

Self Test Answers

1	FALSE:	5 * 7 > 35
2	TRUE:	5 * -7 > -36
3	TRUE:	(3 > 5 - 1) OR (7 < 5 * 2)
4	TRUE:	(27 MOD 5 = 2) AND (27 DIV 5 = 5)
5	TRUE:	(5 > 3) XOR (7 = 7) AND (9 >= 10)
6	FALSE:	(10 > 2) AND ('10' > '2')
7	TRUE:	NOT (11 + 7 < 15) OR ('Great' > 'Greater') AND ('Less' < 'Lesser')
8	TRUE:	TRUE OR FALSE = TRUE
9	TRUE:	NOT (X AND TRUE OR FALSE) = (NOT X OR FALSE) AND TRUE

Explanation: Given that X is Boolean, let’s look at each possibility:

Suppose that X is TRUE:

NOT (TRUE AND TRUE OR FALSE) = (NOT TRUE OR FALSE) AND TRUE

NOT (TRUE OR FALSE) = (FALSE OR FALSE) AND TRUE

NOT TRUE = FALSE AND TRUE

FALSE = FALSE Yes, this is TRUE

Suppose that X is FALSE:

NOT (FALSE AND TRUE OR FALSE) = (NOT FALSE OR FALSE) AND TRUE

NOT (FALSE OR FALSE) = (TRUE OR FALSE) AND TRUE

NOT FALSE = TRUE AND TRUE

TRUE = TRUE Yes, this is also TRUE

Chapter 8
The IF and EXIT Statements

In this chapter, we review two commonly used statements: IF and EXIT.

8.1 Conditional Statements and Boolean Expressions

Once again, to ensure that the terminology is defined the same way for all students, we offer these definitions.

Conditional Statement

This statement tests a condition and executes one or more other statements based on this condition. The IF Statement is the most commonly used conditional statement when there are only two possible values for the condition: either True or False.

Boolean Expression

This expression results in a Boolean value: either True or False. It can be a Boolean variable or constant, a relational expression, or a logical expression. Note that the terms, "Boolean expression" and "logical expression" can be used interchangeably for most purposes. However, we use Boolean expression when we want to emphasize that either a relational expression or a more general logical expression can be used.

8.2 The IF Statement

The IF Statement is used when you want a specific statement to only be executed if a condition is True. It can then be used to execute another statement if the condition is False. There are two ways of writing an IF statement, depending upon the situation.

IF - THEN Syntax

The following syntax is used to execute a statement only if a condition is True:

IF <Boolean expression> THEN <statement>

A True or False condition is stated using a Boolean expression. If the Boolean expression evaluates to True, the statement following the reserved word, THEN, is executed. If the Boolean expression evaluates to False, the statement following THEN is not executed.

For example, if you wanted to test an integer value to see if it was negative and if so, to make it positive, you could use the following IF statement:

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

The relational expression, Amount < 0, is evaluated. If it is True, then the assignment statement is executed and the Amount becomes positive. If it is False, then the assignment statement is not executed and the Amount remains positive.

Note how the above IF statement was written. The statement that is to be executed if the relational expression is True is placed below the IF – THEN line and indented by two characters. Your program will work with or without this indent and it would work even if this statement appeared on the first line, right after the THEN. By convention, we write it using two lines and indenting the second line. Using this technique will make your code much easier to read.

IF-THEN-ELSE Syntax

Remember, there were two ways of writing an IF statement. The following syntax is used to execute one statement if a condition is True and to execute a different statement if that condition is False:

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

A True or False condition is stated using a Boolean expression. If the Boolean expression evaluates to True, the statement following the reserved word, THEN, is executed. If the Boolean expression evaluates to False, the statement following the reserved word, ELSE, is executed. In this type of IF statement , either of the two statements will be executed, but not both.

For example, if you wanted to find the unit price of an item, you might divide the total price by the quantity. However, if the quantity was zero and you did the division, it would result in an error in your code. To prevent this, you would want to test the quantity first. However, even if the quantity is zero, you still want to give the unit price a valid value. Thus, you might write:

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

The relational expression, Quantity <> 0, is evaluated. If it is True (quantity is not equal to zero), the total price is divided by the quantity and the value assigned to the unit price. If it is False (quantity is equal to zero), the unit price is set to zero.

Again, note the way that the IF statement was written above. The two statements that are optionally executed are indented by two spaces. However, the ELSE reserved word is aligned with the IF reserved word. Even though the program works the same regardless of any spaces or new lines, this is a helpful, visual cue as to which ELSE goes with which IF and even helps you to determine which condition triggers the ELSE (just look straight up to the preceding IF).

Another thing to notice is that the first assignment statement does not have a semicolon following it. This is because we are still in the middle of the IF statement. If we were to put the semicolon (statement separator) after the first assignment statement, the system would think that a new statement was coming up. However, the next word after that assignment statement is ELSE, and there is no such thing as an ELSE statement; it is part of the IF statement. A syntax error would result when compiling.

8.3 The EXIT Statement

Normally, code in a trigger executes from the top to the bottom and then control returns to the object that called this trigger, if there is one, or back to the user. If for some reason you do not want to execute the rest of the trigger code, you can use the EXIT statement. When the EXIT statement is executed, the rest of the trigger code is skipped.

The EXIT statement is often used with the IF statement to stop executing trigger code under certain conditions. Suppose that in the above situation, you decided to skip the rest of the trigger code if the Quantity was equal to zero. In this case, you might write:

```
IF Quantity = 0 THEN
  EXIT;

UnitPrice := TotalPrice / Quantity;
```

The relational expression, Quantity = 0, is evaluated. If it is True, then the EXIT statement is executed, which skips the rest of the trigger code. If it is False, the EXIT statement is not executed and therefore the next statement executed will be the assignment statement following the EXIT. Note that this assignment statement is not indented, because it is not part of the IF statement.

8.4 Performing Calculations Using These Statements

We are going to expand on the form that we created in the last lesson to make it do more things. Please refer to the previous lesson if you have any questions on how to create controls, move them, or set their properties.

Adding Additional Variables

For this example, we need to add some variables.

Go into the Object Designer, select Form 95100 and click **Design**. Add the following global variables to the current variable list by selecting **View, C/AL Globals** on the Menu Bar:

```
Quantity     Integer
UnitPrice    Decimal
TotalSales   Decimal
TotalCredits      Decimal
GrandTotal   Decimal
```

Also, change the Data Type of the variable named "Result" from Boolean to Decimal.

Modifying Existing Controls

The existing controls will need some modification so that they will show us the values of two of the new variables.

Close the Globals window and click on the first Text Box, the one whose label says "First Value". Display the Properties window by pressing the Properties button on the Tool Bar. Change the following properties as indicated:

```
Caption      Quantity
SourceExpr   Quantity
```

Click on (or "select") the second Text Box, the one whose label says "Second Value". Change the following properties as indicated:

```
Caption      Unit Price
MinValue     0
SourceExpr   UnitPrice
```

Adding New Controls

We need three new Text Boxes to access the values of the remaining variables and two command buttons to perform the calculations.

Display the Toolbox window, by pressing the Toolbox button on the Tool Bar. Make sure the Add Label tool is selected and add three more Text Boxes with labels to the form. Then, set the properties on each of the three new Text Boxes as follows:

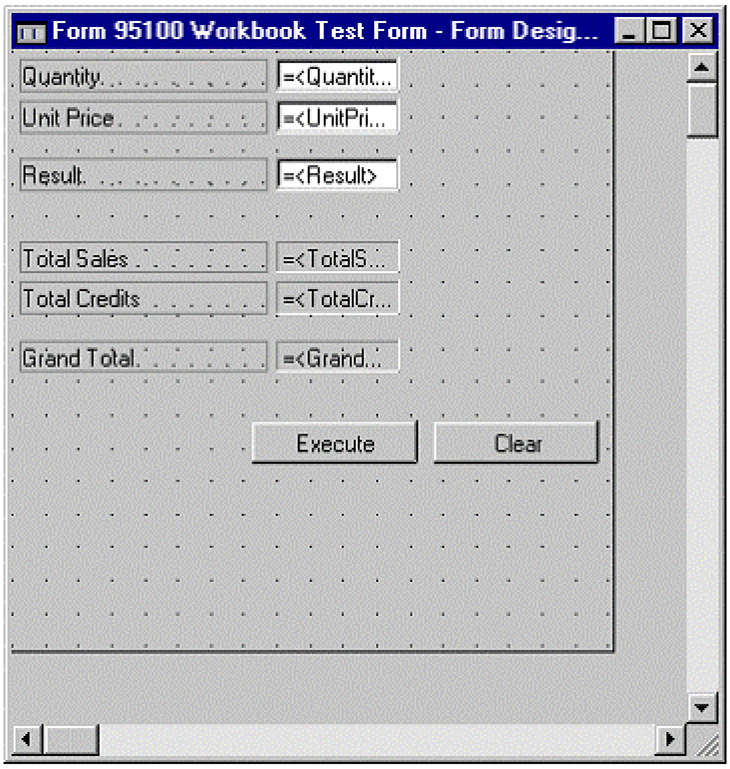
```
First Text Box:
Editable    No
Focusable   No
Caption     Total Sales
SourceExpr  TotalSales

Second Text Box:
Editable    No
Focusable   No
Caption     Total Credits
SourceExpr  TotalCredits

Third Text Box:
Editable         No
Focusable        No
Caption          Grand Total
SourceExpr       GrandTotal
```

Then, deselect the Add Label tool, select the Command Button tool and add one more Command Button to the form. Set its Caption property to "Clear".

Now, using the techniques discussed in the previous lesson, drag the Text Box and the Command Button controls so that the form looks something like this:



Adding New Trigger Code to the Command Buttons

The command buttons require coding to perform our task.

Select the Execute Command Button and press the C/AL Code button on the Tool Bar. Remove the existing code in the OnPush trigger and replace it with the following code. Feel free to use the F5 key to display the Symbol Menu in order to select the variable names, rather than typing them in (for more information on this, review Lesson 4):

```
IF Quantity = 0 THEN
  EXIT;

Result := Quantity * UnitPrice;

IF Result < 0 THEN
  TotalCredits := TotalCredits + Result
ELSE
  TotalSales := TotalSales + Result;

GrandTotal := GrandTotal + Result;
```

The three totals (TotalSales, TotalCredits and GrandTotal) are updated using a standard programming technique called "incrementing". The new value of the total is set to the current value of the total plus the new amount, called the "increment". In the above code, the new GrandTotal is set to the current

GrandTotal plus the value of Result.

When executed, this trigger code does the following: It first checks the Quantity. If Quantity is equal to zero, the rest of the trigger code is skipped; otherwise, Result is set to the product of UnitPrice and Quantity. The value of Result is now tested. If it is less than zero, then TotalCredits is incremented. If it is greater than or equal to zero, TotalSales is incremented. GrandTotal is incremented regardless of the value of Result.

Select the Clear Command Button and display the C/AL Editor again. Add the following code to the OnPush trigger:

```
Quantity := 0;
UnitPrice := 0;
Result := 0;
TotalSales := 0;
TotalCredits := 0;
GrandTotal := 0;
```

When this code is executed, all of the variables are set back to zero.

Setting Data

We now need to test this code.

Exit the Form Designer, save and run the new form. Fill in the Unit Price and Quantity boxes and press the Execute button. Then set the Quantity to another value and press the Execute button again. What happens?

Try entering a negative value in the Quantity (for example, -4) and pressing the Execute button. What happens this time?

What happens if you enter a negative value in the Unit Price? Why does this happen?

What happens when you press the Clear button?

Hand Execution of the Trigger Code

If you do not understand what is happening when you press the Execute button, try hand-executing your code. To do this, write down the starting values of the variables on a slip of paper and for each line of code, check off when it is executed and write the resulting values of the variables down.

Here is an example of how to hand-execute this code. Note that, for this example, the Execute button's code has already been run several times as TotalSales, TotalCredits and GrandTotal have values other than zero assigned to them.

```
IF Quantity = 0 THEN
  EXIT;
Result := Quantity * UnitPrice;
IF Result < 0 THEN
  TotalCredits := TotalCredits + Result
ELSE
  TotalSales := TotalSales + Result;
GrandTotal := GrandTotal + Result;
```

Step 1: ExistingValues Before Execution:

Quantity	UnitPrice	Result	TotalSales	TotalCredits	GrandTotal
-5	2.50	120.50	357.50	-7.00	350.50

Step 2: Test Quantity:

```
IF Quantity = 0 THEN
```

Quantity	UnitPrice	Result	TotalSales	TotalCredits	GrandTotal
-5	2.50	120.50	357.50	-7.00	350.50

Note that no variables are changed here, only tested. The result is that the EXIT statement is not executed.

Step 3: Calculate Result:

```
Result := Quantity * UnitPrice;
```

Quantity	UnitPrice	Result	TotalSales	TotalCredits	GrandTotal
-5	2.50	-62.50	357.50	-7.00	350.50

The new value of Result is negative 62.50.

Step 4: Test Result:

```
IF Result < 0 THEN
```

Quantity	UnitPrice	Result	TotalSales	TotalCredits	GrandTotal
-5	2.50	-62.50	357.50	-7.00	350.50

Again, no variables are changed at this time. The value of Result is being tested and the next statement that will run is being determined. Since Result is < 0, TotalCredits will be incremented, while the statement that increments TotalSales will be skipped.

Step 5: Increment TotalCredits:

```
TotalCredits := TotalCredits + Result
```

Quantity	UnitPrice	Result	TotalSales	TotalCredits	GrandTotal
-5	2.50	-62.50	357.50	-69.50	350.50

The new value of TotalCredits is its current value plus Result.

Step 6: Increment GrandTotal:

```
GrandTotal := GrandTotal + Result;
```

Quantity	UnitPrice	Result	TotalSales	TotalCredits	GrandTotal
-5	2.50	-62.50	357.50	-69.50	288.00

The new value of GrandTotal is its current value plus Result.

The check marks below show which lines were run in this example.

```
✓ IF Quantity = 0 THEN
    EXIT;
✓ Result := Quantity * UnitPrice;
✓ IF Result < 0 THEN
✓   TotalCredits := TotalCredits + Result
  ELSE
    TotalSales := TotalSales + Result;
✓ GrandTotal := GrandTotal + Result;
```

8.5 Self Test

1. Using the Workbook Test Form that you have been developing, add the ability to accumulate the total quantity sold, the total quantity credited and the grand total quantity.

2. Again, using the same form, add two counters that can be incremented by 1 every time you press the Execute button and are set to zero whenever you press the Clear button. One counter should be used to count the number of sales (Quantity > 0) and the other should count the number of credits (Quantity < 0). Also, when the Execute button is pressed, use these counters to calculate the average quantity sold and the average quantity credited.

Self Test Answers

1 Add the ability to accumulate the total quantity sold, the total quantity credited and the grand total quantity.

Add the appropriate variables into the Global Variables (make sure they are all Integer variables, like Quantity). Add controls to the form to display the values of these variables. Add the following code at the end of the existing OnPush code for the Clear button:

```
TotalQtySold := 0;
TotalQtyCredited := 0;
GrandTotalQty := 0;
```

Using the names of the variables that you created, add the following code at the end of the existing OnPush code for the Execute button:

```
IF Quantity < 0 THEN
  TotalQtyCredited := TotalQtyCredited + Quantity
ELSE
  TotalQtySold := TotalQtySold + Quantity;
GrandTotalQty := GrandTotalQty + Quantity;
```

2 Add two counters that can be incremented by 1 every time you press the Execute button and are set to zero whenever you press the Clear button. One counter should be used to count the number of sales (Quantity > 0) and the other should count the number of credits (Quantity < 0). Also, when the Execute button is pressed, use these counters to calculate the average quantity sold and the average quantity credited.

Create two additional global integer variables (AvgQtySold, AvgQtyCredited). Add controls to the form to display the values of these variables. Add the following code to the end of the OnPush code for the Clear button:

```
SalesCounter := 0;
CreditCounter := 0;
AvgQtySold := 0;
AvgQtyCredited := 0;
```

Add the following code to the end of the Execute button's OnPush code:

```
IF Quantity < 0 THEN
  CreditCounter := CreditCounter + 1
ELSE
  SalesCounter := SalesCounter + 1;
IF SalesCounter <> 0 THEN
  AvgQtySold := TotalQtySold DIV SalesCounter;
IF CreditCounter <> 0 THEN
  AvgQtyCredited := TotalQtyCredited DIV CreditCounter;
```

Note that the DIV operator is used here. If division (/) was used and the result contained a decimal, the result could not be assigned to an integer variable.

Chapter 9
Compound Statements and Comments

This chapter reviews the use of compound statements to perform multiple assignments. It explains the use of BEGIN and END and nested IF statements. The use of indentation is shown in order to make the code easier to read. It also reviews adding comments to the C/AL code.

9.1 Compound Statements and Comments

Below are the definitions of the two terms that we will be using in this chapter.

Compound Statement

This statement enhances the capabilities of the conditional statement. Remember that the conditional statement can only run one statement if the condition is found True. If there is an ELSE built in to the statement then it will run one statement if the condition is False. However, many times we will need to run more than one statement for either or both conditions. This is where the compound statement comes in.

The compound statement actually consists of multiple statements. For example, in an IF statement, if you wanted to perform two or more assignments when a condition tested to True, you would place a compound statement after the THEN. Normally after a THEN, you are allowed to place only one statement, However, if that one statement is a compound statement, you can perform multiple assignment statements in it.

Comment

A comment is a description that you place in your code, possibly to explain the code, or to document who did a modification or why it was done. However, since it is a description and not an instruction, we do not want the compiler to read it. Using a special syntax, we inform the compiler to ignore what is written in the comments and therefore, it does not attempt to translate them.

9.2 Compound Statement Syntax with BEGIN and END

In order to use Compound Statements, we need to understand their syntax.

Compound Statement Syntax

The syntax of a compound statement is as follows:

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

The braces ({ }) indicate that whatever is included inside can be repeated zero or more times. In other words, there may not be a second statement at all, or there may be two or five, however many are needed. The curly bracket is also known as a "brace".

IF-THEN Compound Statement

A compound statement can be used wherever, syntactically, a single statement can be used. For example, if you wanted to first calculate a unit price from an extended price, then add the extended price to the total price, but only if the quantity was not zero, you could write the following IF statement:

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

Note the indentation that is used with the compound statement. The statements in a compound statement are always indented two spaces from the BEGIN and the END, which are aligned with each other. If this rule were strictly followed, along with the normal indentation rules for IF statements, the result would have been:

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

However, to prevent this double indentation, while still making the code easy to follow, there is a special indentation rule for IF statements. In an IF statement, the BEGIN is placed 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

Let's include the ELSE clause. Suppose that if the Quantity was zero, you not only wanted to set the unit price to zero, but you also wanted to skip the rest of the trigger. Using the "normal" indentation rules, you would expect to see:

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

However, we also have special indentation rules for use with ELSE clauses. To reduce the number of lines of code (keeping more on the screen at once) and still make the code easy to read, we allow two changes. First, the ELSE appears on the same line as the END from its corresponding IF. Secondly, the BEGIN is on the same line as the ELSE. Thus the Navision standard indentation for the above IF statement would be:

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

9.3 Compound Statements using Nested IF Statements

The statement in the THEN or the ELSE clause of an IF statement could be another IF statement instead of an assignment. When this occurs, the second IF statement is called a "nested" IF statement. These can be complicated to understand, especially when trying to figure out which IF statement goes with an ELSE.

Nested IF Statement Confusion

The rule is that the ELSE goes with the closest IF above it that does not already have an ELSE. Here, indentation rules can really help when trying to understand code. For example, look at the following code that is not indented:

```
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;
```

It is hard to tell what circumstances would cause Sales to be reduced by Amount without diagnosing this code. Even worse, suppose it was indented incorrectly. This could be downright misleading:

```
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;
```

Proper Nested IF Statement Indentation

However, using the rule that the ELSE goes with the closest IF without an ELSE and the standard indentation rules that were covered in the above sections, you could rewrite it like this:

```
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;
```

Note that in all of these examples, the code would execute the same, since the compiler ignores all spaces and new lines. However, in the last example, it is much easier for the programmer to tell what is going on.

Also, note the importance of having meaningful names for your variables. Suppose the same code was indented correctly, but the variables' names were not meaningful, like this:

```
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 is often how well your code documents itself. Through proper indenting, use of meaningful variable names, using Booleans to designate yes or no choices and using the Option type and option constants rather than integers to designate selections with more than two choices, your code can be mostly self-documenting. Just reading this type of code will tell you what it means. Sometimes, however, it is useful to add additional documentation to your code. This brings us to the next topic.

9.4 Adding Comments to Code

There are two different syntaxes for Comments. Either one may be used anyplace. However, there are some advantages of one over the other, depending upon the situation.

Single Line Comment

The first syntax is a single line comment; that is, it comments out a single line in the code. To “comment out” means that the entire section of code becomes a comment, which will be ignored by the compiler. It is created by placing two slashes next to each other (//) on a line. Everything on that line after those two slashes is considered a comment and is ignored by the compiler. Here are some examples:

```
// 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 program line being used entirely for a comment. The other two show that you can have some code on a line and then follow it with a Comment. In each case, the compiler ignores anything following the two slashes on that line.

Block of Comments

The second syntax is a block of comments. If a brace ({) is inserted in the code, the compiler will ignore it and everything following it until the matching brace (}) is found. The compiler ignores even new lines in this case. Here is an example of a block of Comments:

```
{The following code is used to calculate the Unit
Price.  Note that we compare the Quantity to zero in
order to prevent division by zero.  In this case, we
set the Unit Price to zero.}

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

Nested Comments

One of the common uses of a block of comments is when you are tracking down problems. A brace is inserted at the beginning of a section of code and a matching brace is inserted at the end of the section of code. This will cause the entire section of code to comment out. When you comment out a section of code, it allows you to concentrate your efforts on the remaining parts of the code, or eliminate a part of the code as the cause of a problem.

But what if there is already a set of braces in the code, indicating a comment? This is not a problem.

Comment blocks can be nested. When the compiler reaches a brace, it will treat everything as a comment until it reaches the matching brace. Thus, in your trigger code, a closing brace must match every opening brace. Here is an example of a nested comment:

```
{The following code temporarily removed on 8/15/01...

{The following code is used to calculate the Unit
Price.  Note that we compare the Quantity to zero to
prevent division by zero.  In this case, we set the
Unit Price to zero.}

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

Note that when this section of code is run, only the line that sets the unit price to zero is actually executed. The rest are skipped.

9.5 Coding with Compound Statements and Comments

After the exercises in Lesson 8, along with the Self-Test in the same lesson, the OnPush trigger of the Execute button has quite a bit of code in it. However, a lot of it can be consolidated, now that we can use compound statements.

Go into Form 95100 and modify the code to take full advantage of compound statements. The result might look something like this:

```
IF Quantity = 0 THEN
  EXIT;
Result := Quantity * UnitPrice;
IF Quantity < 0 THEN BEGIN
  TotalCredits := TotalCredits + Result;
  TotalQtyCredited := TotalQtyCredited + Quantity;
  CreditCounter := CreditCounter + 1;
  AvgQtyCredited := TotalQtyCredited DIV CreditCounter;
END ELSE BEGIN
  TotalSales := TotalSales + Result;
  TotalQtySold := TotalQtySold + Quantity;
  SalesCounter := SalesCounter + 1;
  AvgQtySold := TotalQtySold DIV SalesCounter;
END;
GrandTotal := GrandTotal + Result;
GrandTotalQty := GrandTotalQty + Quantity;
```

Because this code is short and easy to understand, the programmer may not normally add comments. However, just for the exercise, try adding some comments, both a comment block and several single line comments. Try compiling to make sure everything is still correct.

9.6 Self Test

1 In the following set of statements, draw a line from each ELSE to its matching IF statement:

```
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;
```

2 After executing the following set of statements, what value does variable A5 have?

```
{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;
```

Self Test Answers

1 Here is the same set of statements, indented correctly, with lines drawn from each ELSE to its corresponding IF, so that you can see the relationship.

```
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;
```

2 Below are the same statements that have had the commented sections removed and are indented correctly.

```
A5 := 7;
B1 := FALSE;
B2 := TRUE;
A1 := 11;
A2 := 2 * A5;
IF (A1 < A5) OR B1 THEN
  A5 := 3 * A5
ELSE
  A5 := A1 + A5;
```

```
A5 = 18
```

Chapter 10
Arrays

This chapter examines arrays – what they are and when and how to use them in C/AL code.

The following topics will be covered:

10.1 Array Definitions

Arrays are special variables that have more functionality than the variables that we have discussed to this point.

Review of Simple Data Types and Variables

Simple data types were covered in Lesson 2; they were defined as types of data that only have a single value. A simple variable is a variable defined as a simple data type.

Complex Data Types and Variables

A complex data type is a type of data with multiple values. A complex variable is a variable with a complex data type, that is, a variable that has multiple values.

Array

An array is a complex variable that actually holds a group of variables. This group is defined all at once with a single identifier and a single data type. For example, we could create an array variable with the following identifier and data type:

- Identifier: QuantityArray
- Data Type: Integer

Element

This term is a single variable in an array. An array is made up of one or more elements, although it is possible to create an array with one element, this is rarely done, as it would be no more useful than a simple variable. All elements in an array have the same data type, whatever the array was defined to be. The above array needs to have the number of elements defined.

- Elements: 5

Index

An index is used to refer to a single element in an array. In order to access a single element in an array, you need to use both the array variable name and a number, the index, to indicate the desired array element. To access the fourth element in the above example, you would specify:

· QuantityArray[4]

Dimension

An array can have one or more dimensions. The easiest array is a one-dimensional array, which is what we have defined, by default, above. This array has only elements in one dimension. This can be compared to having only elements on the x-axis of a graph.

In the above example, there are a grand total of 5 elements (one line, or dimension, of 5 defined elements.)

If we define the array as two dimensions, it then has a grand total of 25 elements (5 elements in one dimension times 5 elements in the second dimension.) Think of this as having a two dimensional graph, using the x-axis and y-axis. Each line has 5 elements (1 through 5), but provides a combination of 25 different points (using only integers).

The mathematical formula for giving the grand total of elements in an array is the defined element number raised to the dimension number. Arrays can have up to 10 Dimensions in C/AL.

In order to use a particular element in a multi dimensional array, you would specify an index value for each dimension. Therefore, there will be the same number of index values as the defined dimensions. Remember that these values must be integers.

10.2 Array Syntax

Remember that an array is still a variable. The only difference between calling a simple variable and a complex one is adding the element index.

Variable Syntax

When you want to refer to any variable or an array as a whole, you use its identifier. For example, there is a built-in function called CLEAR, which has the following syntax:

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

The CLEAR function clears the variable you use as the parameter. If the variable is a numeric type, the CLEAR function sets it to zero. If the variable is a string type, its sets it to the empty string. If the variable is an array, then each element in the array is set to its own cleared value. Thus, if you had a one dimensional array named SaleAmount whose data type was set to decimal, the following function call would set all elements in SaleAmount to zero.

```
CLEAR(SaleAmount);
```

Array Element Syntax

When you want to refer to a single element of an array, you must refer to it with its identifier and its index, according to the following syntax:

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

The brackets ([]) above are literal brackets, while the braces ({}) indicate that whatever is included in them can be repeated zero or more times. A one - dimensional array will require a single index expression, while a two dimensional array will require two index expressions, separated by a comma. Each index expression must result in an integer value when evaluated. Using the same SaleAmount array described above, if you wanted to set the fifth element to zero, you would use the following assignment statement:

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

How to Think About Arrays

An easy way to think about a one-dimensional array, is to think of a row of boxes, like this:

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

Here, we have a one-dimensional array, whose identifier is Box and which has 8 elements. Each element of the array is a single box. The fourth element of the array contains the value of 17. If we want to refer to the fourth element of the array, we would refer to Box[4].

It is also easy to think about a two-dimensional array, if you think of it like a 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]

Here, we have a two-dimensional array, whose identifier is Box and that has a total of 30 elements, broken up into 5 rows of 6 elements. The first dimension can be thought of as the row number, while the second dimension is thought of as a column number. In order to identify a specific element, you must identify both the row and the column. For example, if we wanted the value of the element in the 4 row, 6th column, we would refer to Box[4,6] and the value we would pick up would be 24.

Once you get to 3 or more dimensions, it starts getting harder to visualize. You have to start thinking in terms of the index being a list of criteria to match. Fortunately, there are few reasons to have a more than one-dimensional array in Navision Solutions, although up to 10 are allowed. For the rest of this lesson, we will limit ourselves to one-dimensional arrays.

10.3 The Power of Arrays

If all you could do with an array is access a particular element, like this...

```
SaleAmount[5]
```

....then it would not be very useful. At this point, creating an array with 10 elements only saves the developer from creating 10 different variables. Of course, there is much more that the developer can do with an array.

Using Expressions as an Index Value

The true power of an array is in the fact the index value can be an expression:

```
SaleAmount[Counter]
```

This allows the element you are addressing to be determined at run-time. In the above example, the element is determined by whatever value Counter has. This can be very useful as is explored in the exercise below, as well as in the following chapter, where we will talk about repetitive statements.

However, this does bring up a possible problem. How do we know, at run-time, whether the value of the index expression is a valid element number? If an array has been defined to have 8 elements and we refer to element number 12, it will result in a run-time error.

To stop this, we will often use an IF statement, to test the value of the variable before we use it to index an array. For example:

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

ARRAYLEN Function

However, the problem with this method is that the array might actually be defined to have 7 elements and we could never tell that this line was not correct until a run-time error occurred. To address this problem C/AL has a built-in function called ARRAYLEN, with the following syntax:

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

The "Result :=" above indicates that the ARRAYLEN function results in a value, in this case of type Integer. Using the above syntax, the result will be the number of elements in the array variable used as the parameter. For example, if a one-dimensional array has been defined to have 8 elements, the function below will have the integer value of 8:

```
ARRAYLEN(SaleAmount)
```

Note: ARRAYLEN can also be used for more than one-dimensional arrays. If you are interested, look at the Help for this function in C/SIDE.

The advantage of using ARRAYLEN is that if you later go back and change the defined length of an array, you do not have to back through your code to update references to the defined length. Thus, in our first example above, we could test to see if we have a valid index value like this:

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

Since the number 8 is never used, then nothing here says the array must have exactly 8 elements. If it turns out the array is actually defined to have 7 elements, this code will still work perfectly.

10.4 Strings as Arrays of Characters

A string variable (variable of type text or code) can be thought of, for some purposes, as an array of characters. Because of this, C/AL allows access to each character as an element in an array.

Characters as Elements

Each element is considered a variable of type char. For example, if you ran the following code...

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

...the resulting message would be "Walk in the dark".

Note that since char is a numeric type, you can use the ASCII codes for characters when using strings this way.

For example, if you imported a text line, you could check for a tab character like this:

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

Please note, however, that the length of a string cannot be accessed in this manner. No matter which characters you read or set, the length of the string remains the same. The elements beyond the string's length are considered undefined.

In our first example, if we had set the 25th element (rather than the 13th) to 'd', the message displayed would have said, "Walk in the park". There would have been no change, since the 25th character was beyond the length of the string.

Similarly, in the second example, if idx had been greater than the length of Str, it still might find a tab character there, but it would just be garbage, not actually part of Str.

Also, if you want to know the number of elements in a string being used as an array of characters, you must use the MAXSTRLEN function. The ARRAYLEN function will not work for this purpose.

10.5 Using Arrays

We need to understand how to create and use arrays in C/SIDE.

Creating an Array Variable

Go into the Object Designer, select Form 95100 and click **Design**. Add the following global variables to the current variable list, by selecting **View, C/AL Globals** on the Menu Bar:

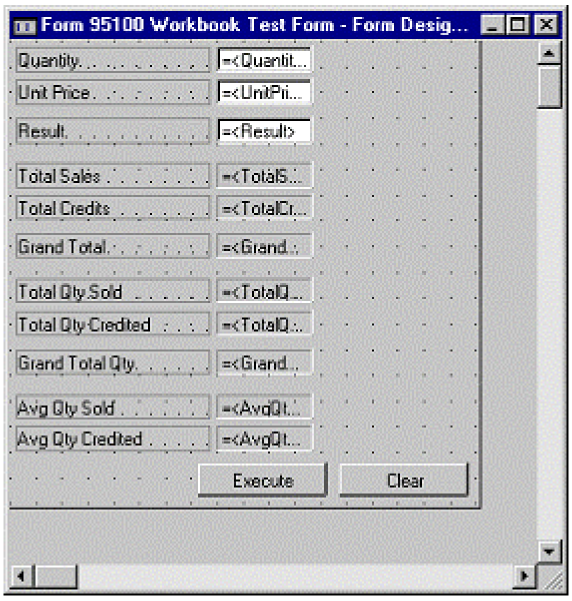
```
Counter           Integer
Amount            Decimal
```

With the cursor on the same line as the Amount variable, display the Properties window by pressing the Properties button on the Tool Bar. Change the Dimensions property to a value of 10. This means that Amount will be a one - dimensional (since only one number was entered) array variable, of type decimal, with a maximum value of the first (and only) dimension being 10, which means that there will be 10 elements.

Note: We will not do this now, but if you wanted to create a two-dimensional array, you would put two numbers in the Dimensions property, separated by a semicolon (;). For example, to create the Box array from the above lesson, you would put 5;7 in the Dimensions property. This would create a two dimensional array, with the maximum value of the first dimension being 5 and the maximum value of the second dimension being 7, which means that there will be 35 elements.

Adding Controls to View the Array

Close the Globals window. By now, your form should look something like this:



Form 95100 Workbook Test Form - Form Desig...

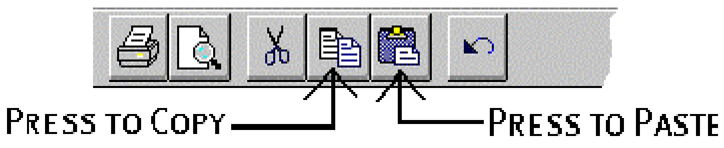
Quantity	=<Quantit...
Unit Price	=<UnitPri...
Result	=<Result>
Total Sales	=<TotalS...
Total Credits	=<TotalCr...
Grand Total	=<Grand...
Total Qty Sold	=<TotalQ...
Total Qty Credited	=<TotalQ...
Grand Total Qty	=<Grand...
Avg Qty Sold	=<AvgQt...
Avg Qty Credited	=<AvgQt...

Display the Toolbox window by pressing the Toolbox button on the Tool Bar.

Make sure the Add Label tool is not selected and add a Text Box (with no label) to the form, just to the right of Result. Then, set the properties on this new Text Box as follows:

```
Editable     No
Focusable    No
BlankZero    Yes
SourceExpr   Amount[1]
```

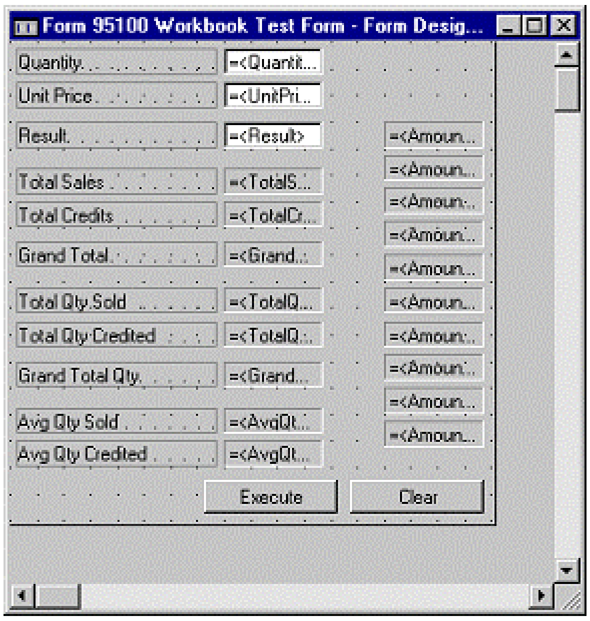
Now, making sure that the new control is still selected (if not, click on it), press the Copy button on the Tool Bar:



Now, press the Paste button on the same Tool Bar nine times. Each of the new controls should appear below the previous control, so you end up with a list of 10 controls.

Now select each control in turn, go to its properties and change the SourceExpr so that each control references the corresponding element in the Amount array. For example, the second control ought to be changed to Amount[2], the third control to Amount[3] and so on, until the tenth control is changed to Amount[10].

By now, your form should look something like this:


Form 95100 Workbook Test Form - Form Desig...

Quantity	=<Quantit...	
Unit Price	=<UnitPri...	
Result	=<Result>	=<Amoun...
		=<Amoun...
Total Sales	=<TotalS...	=<Amoun...
Total Credits	=<TotalCr...	=<Amoun...
Grand Total	=<Grand...	=<Amoun...
Total Qty Sold	=<TotalQ...	=<Amoun...
Total Qty Credited	=<TotalQ...	=<Amoun...
Grand Total Qty	=<Grand...	=<Amoun...
		=<Amoun...
Avg Qty Sold	=<AvgQt...	=<Amoun...
Avg Qty Credited	=<AvgQt...	

Add Code to the Clear Button

Select the Clear button and access its OnPush trigger code. Let's set all of the variables to zero.

Rather than setting each element individually to zero, we set the entire array to zero using the CLEAR function. Not only does this save a lot of code (consider if we had said that Amount had 1000 elements), but it also does not force us to know how many elements there are. Add the following two lines to set both of the new variables to 0:

```
Counter := 0;
CLEAR(Amount);
```

Add Code to the Execute Button

Select the Execute button and access its OnPush trigger code. At the end, add the following lines of code (with a comment):

```
// Display results in Amount column on form
Counter := Counter + 1;
Amount[Counter] := Result;
```

Note what we are doing here:

· The first time you press the Execute button, Counter will be incremented to 1 and the element Amount[1] will be set to the result. Amount[1] is displayed in the first of the new controls that we added.

· The second time you press the Execute button, Counter will be incremented to 2 and the element Amount[2] will be set to the result. Amount[2] is displayed in the second of the new controls that we added.

Save and run the form. Add some Unit Prices and some Quantities, pressing the Execute button after each. Watch what happens. Now press the Clear button. What does it do?

Now, enter more Unit Prices and Quantities and continue to press the Execute button until you have filled the Amount column. Then press Execute one more time. What is the result? Why?

Let's go back and fix this last problem. Go into the Execute button's OnPush trigger and modify those last lines we just added so that they now look like this:

```
// Display results in Amount column on form
Counter := Counter + 1;
IF Counter <= ARRAYLEN(Amount) THEN
  Amount[Counter] := Result;
```

Save and run this form again. What happens when you press the Execute button more than 10 times now? We have now solved this problem. We need to keep this problem in mind anytime an array is used.

10.6 Self Test

In each of these examples, explain if this is an allowable application for arrays.

1 We have a list of students, numbered from 1 to 99. Each student takes a test. We 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, etc.

2 We have a list of students, numbered from 1 to 99. Each student takes a test. We want to create a two dimensional array with two rows of 99 columns. In the first row, we would put the corresponding student's name, with student 1's name in element 1,1, student 2's name in element 1,2, etc. In the second row, we would 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, etc.

3 We create an array containing the number of households in each ZIP code. There will be one array element for each 5 digit ZIP code and each element will contain the number of households in that ZIP code. The number of households that have a ZIP code of 30071 will go into element 30071, etc.

4 We create an array containing the number of households in each state. There will be one array element for each 2-character postal state code and each element will contain the number of households in that state. The number of households in Georgia will go into element 'GA', etc.

Self Test Answers

1 We have a list of students, numbered from 1 to 99. Each student takes a test. We 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, etc.

This will work.

2 We have a list of students, numbered from 1 to 99. Each student takes a test. We want to create a two dimensional array with two rows of 99 columns. In the first row, we would put the corresponding student's name, with student 1's name in element 1,1, student 2's name in element 1,2, etc. In the second row, we would 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, etc.

This will not work. Every element in an array must have the same data type. In this case, the student names are text and their scores are integers. For this to work, you must use two separate arrays.

3 We create an array containing the number of households in each ZIP code. There will be one array element for each 5 digit ZIP code and each element will contain the number of households in that ZIP code. The number of households that have a ZIP code of 30071 will go into element 30071, etc.

This will work. The maximum integer is over two billion, while the maximum ZIP code is less than one hundred thousand. Also, the maximum number of elements in a single array is one million. We are well within the limits.

4 We create an array containing the number of households in each state. There will be one array element for each 2-character United States postal state code and each element will contain the number of households in that state. The number of households in Georgia will go into element 'GA', etc.

This will not work. Although array elements can be of any type, array index values must be integer. In this case, the array index is text or code.

Chapter 11
Repetitive Statements

This chapter covers the repetitive statements FOR, WHILE...DO and REPEAT...UNTIL.

This chapter includes:

11.1 Repetitive Statments

A repetitive statement is a statement that allows you to execute one or more other statements multiple times. There are several kinds of repetitive statements and they differ on how they determine the number of times their included statement(s) is (are) executed. A repetitive statement is often called a "loop", because when the execution reaches the end of the repetitive statement, it loops back to the beginning of the repetitive statement.

This chapter reviews three different types of repetitive statements, their similarities and their differences.

11.2 The FOR Statement

The FOR statement is used when a statement is to be executed a predetermined number of times. There are two FOR statements.

FOR...TO Statement

The most common, FOR...TO, has this syntax:

```
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 be variables of that same data type. For 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 of type integer. The chart below describes what happens:

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
3.	The control variable is compared to the end value. If it is greater than the end value, the FOR statement is terminated.	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.	

The net result, for this example, is that the Total variable is increased by 2.5 a total of 5 times, therefore increasing it by 12.5. Note that both the start value and the end value are evaluated only one time, at the beginning of the FOR statement.

Note that the control variable's value should not be changed in the FOR loop, as the results of that would not be predictable. Also, the value of the control variable outside of the FOR loop (after it ends) is not defined.

Using BEGIN and END

What if several statements need to be run inside the loop? We then need to use a compound statement by adding BEGIN and END. Here's an example:

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

FOR...DOWNTO Statement

The second FOR statement, the FOR DOWNTO statement, has this syntax:

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

The rules for the control variable, start value and end value are the same as for the other FOR 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, where in the FOR DOWNTO statement, the control variable decreases in value until it is less than the end value. Here is an example and the explanation using an array and a compound statement:

```
FOR idx := 9 DOWNTO 1 DO BEGIN
  TotalSales := TotalSales + Sales[idx];
  NumberSales := NumberSales + 1;
END;
```

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 TO) 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 terminated.	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 decrements 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 9 and then 8, and so on.

11.3 The WHILE...DO Statement

The WHILE loop is used when a statement is to be executed as long as some condition holds true. Here is the syntax of a WHILE statement:

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

This is simpler than the FOR statement. As long as the Boolean expression evaluates to True, the statement, or statements if using BEGIN and END, is executed repeatedly. Once the Boolean expression evaluates to False, the statement is skipped and execution continues with the statement following it.

Here is an example, showing the use of a compound statement:

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

Here is the explanation:

1.	The Boolean expression is evaluated. If it is true, continue with step 2. If not, the	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.	

Note that the Boolean expression is tested before the statement is even executed one time. If it evaluates to False from the beginning, the statement is never executed.

Also, note that the Sales[idx] that is added to Total Sales 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 is what causes this. Also, once the WHILE loop has ended, idx will still refer to the last non-zero element in the Sales array.

11.4 The REPEAT...UNTIL Statement

The REPEAT statement is used when one or more statements are to be executed until some condition becomes true. Here is the syntax of the REPEAT statement:

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

There are several differences between the REPEAT and WHILE statements:

· First, there can be more than one statement between the REPEAT and the UNTIL, even if no BEGIN and END is used.

· Secondly, the Boolean expression is not evaluated until the end, after the statements have already been executed once.

· And third, when the Boolean expression is evaluated, it loops back to the beginning if the expression evaluates to False and terminates the loop if the expression evaluates to True.

Here is an example, which is almost identical to the example used for the WHILE statement:

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

Here is the explanation:

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 is true, the REPEAT statement is terminated. If not, go back to step 1.	Is Sales[idx] = 0?

Note that since 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 will refer to the first Sales which equals zero, rather than the last non-zero Sales as in the WHILE loop.

Also note that the Boolean expression had to be rewritten, since we are now testing for a False condition to continue the loop, rather than a True expression as in the WHILE loop.

11.5 Coding with Repetitive Statements

As you recall from our last chapter, we left our test form so that the first 10 items you added to the list were listed, but from then on nothing showed in the list. We will now change this so that the last 10 items you added to the list will always show.

Adding a FOR Loop

Go into the Object Designer, select Form 95100 and click **Design**.

Add the following global variable to the current variable list by selecting **View, C/AL Globals** on the Menu Bar:

```
idx         Integer
```

Select the Execute button and access its OnPush trigger code. Find the code following the comment that says "Display results in Amount column on form". Modify it so that it looks like this:

```
// Display results in Amount column on form
Counter := Counter + 1;
IF Counter > ARRAYLEN(Amount) THEN BEGIN
  Counter := ARRAYLEN(Amount);
  FOR idx := 2 TO Counter DO
    Amount[idx-1] := Amount[idx];
END;
Amount[Counter] := Result;
```

Note what we are doing here. Once Counter becomes greater than the length of the Amount array, we set it back to be equal to that length. Then, we move each of the elements in the Amount array up by one index, using a FOR statement. Element 2 is moved to element 1, then element 3 is moved to element 2, etc. Finally, element 10 is moved to element 9 and then the FOR loop ends. The new result is then moved into the last element in the array.

Now save and run this form.

Enter some values, press the Execute button, and watch what happens. Until the list is filled, nothing changes, but watch what happens when you have pressed the Execute button more than 10 times.

Sorting – Phase I

Go back into the design of the form and add a new Command Button with its Caption property set to Sort. If you need help doing this, refer back to the exercise in Lesson 7.

Highlight the button and press the F9 key to find the OnPush trigger and add the following code:

```
FOR idx := ARRAYLEN(Amount) DOWNTO 2 DO
  IF Amount[idx] < Amount[idx-1] THEN BEGIN
    TempAmount := Amount[idx];
    Amount[idx] := Amount[idx-1];
    Amount[idx-1] := TempAmount;
  END;
```

For this to compile, you will also need to add a variable called TempAmount of type Decimal. Now exit, compile, save and run this form.

Fill in some values and press the Execute button, entering differing quantities each time until the column is full. Now press the Sort button. What happens?

Press the Sort button again. What happened this time?

Continue pressing the Sort button until nothing further happens. Note that at this point all entries are sorted. Now, examine the above code and answer the following questions. Hand execute the code if necessary.

1 Why in the FOR statement do we only go down to 2 rather than all the way to 1?

2 Why don't we need a BEGIN after the DO in the FOR statement?

3 What is taking place in the 3 lines of the compound statement (between BEGIN and END)?

4 Why is TempAmount used for this?

5 If the entries in the column had been made exactly in the wrong order (highest to lowest), how many times would the Sort button have to be pressed to sort the list?

Sorting – Phase II

Obviously, it would be better if the user could just press the Sort button once and have the list completely sorted. Note, however, that sometimes one press of the button would work, while other times a lot more are needed. A FOR loop would be inappropriate for this, because we do not know in advance how many times it will take.

For this situation, a REPEAT loop would work best. Add a new variable called IsSorted of type Boolean.

Then modify the Sort button's OnPush code as follows:

```
REPEAT
  IsSorted := TRUE;
  FOR idx := ARRAYLEN(Amount) DOWNTO 2 DO
    IF Amount[idx] < Amount[idx-1] THEN BEGIN
      TempAmount := Amount[idx];
      Amount[idx] := Amount[idx-1];
      Amount[idx-1] := TempAmount;
      IsSorted := FALSE;
    END;
UNTIL IsSorted;
```

Note that the IsSorted variable is used as a signal to tell whether we are finished sorting. We set it to TRUE at the beginning of each REPEAT loop and if we ever have to swap two elements because the list is not in order, we set it to FALSE.

At the end of the REPEAT loop, we check to see if it is TRUE. If it is, we know that all of the elements are in order, since none needed to be swapped and therefore we can exit the loop. If not, we repeat the loop again. When we use a Boolean variable as a signal in this way, programmers sometimes call it a "flag".

Also, note that we cannot tell whether the list is sorted unless we actually run through it at least one time. That is why we used a REPEAT here, rather than a WHILE loop.

Save and run this form. Perform the same experiment as before, filling in the table with values and then pressing the Sort button. Note that the list is completely sorted with one press.

Look at the IF statement again. We used the less than (<) operator to compare the elements.

1 What would happen if you changed this to the greater than (>) operator?

2 What would happen if you changed it to the less than or equal (<=) operator? If you are not sure, try it in your code.

Test the case where two elements happen to have the same value. Once you find out, change it back to the less than operator for the next exercise.

Sorting – Phase III

We now have a perfectly good sorting routine. It works quite well for small arrays. But notice that we wrote it so that it could take arrays of any size. What if there were 100 elements, 1,000 elements, or 10,000 elements? We created "nested" loops when we put one loop (the FOR loop) inside another loop (the REPEAT loop). Whenever there are nested loops, there is always a potential for inefficiency, especially as the number of elements increase. The question becomes, how many times will the innermost loop be executed?

For example, in this case, whenever the FOR loop is executed once, the IF statement is executed 9 times, because the length of the array is 10, but we are not going all the way down to 1. If the REPEAT loop has to be executed 9 times, the FOR loop will be executed 9 times and the IF statement will be executed 9 * 9 = 81 times. However, if the array length goes up to 100, the IF statement would be executed 99 times per FOR statement and the FOR statement could be executed by the REPEAT loop up to 99 times. In this situation, the IF statement might be executed up to 99 * 99 = 9,801 times!

We can reduce this substantially, by remembering what happened when we first wrote the FOR loop by itself. Every time we ran it, the lowest element would rise to the top. If you think about it, that means that the first element would never need to be tested again, once the first FOR loop was run. Once the second FOR loop was run, you would never need to test the first two elements again. Each loop needs fewer and fewer tests to do its job. And this is the worst case. Sometimes, when we pressed the sort button, the first 3 elements became sorted and we would really never need to test any of the first 3 again!

Here is how we can take advantage of this. First, create an integer variable called LowestSwitch.

Now, rewrite the sorting routine so it looks like this:

```
LowestSwitch := 2;
REPEAT
  IsSorted := TRUE;
  FOR idx := ARRAYLEN(Amount) DOWNTO LowestSwitch DO
    IF Amount[idx] < Amount[idx-1] THEN BEGIN
      TempAmount := Amount[idx];
      Amount[idx] := Amount[idx-1];
      Amount[idx-1] := TempAmount;
      IsSorted := FALSE;
      LowestSwitch := idx + 1;
    END;
UNTIL IsSorted;
```

Instead of sorting down to 2 (comparing with 1) every time, now the FOR loops says to sort down to LowestSwitch (comparing with LowestSwitch - 1). LowestSwtich starts out as 2, but each time two values are switched (in the THEN clause of the IF statement), it is reset to one more than the element currently being switched. Since the FOR loop works down, at the end, LowestSwitch will be the lowest element to test next time around. Even in the worst case, it will be one higher than the previous value and it might be several higher. Each one higher is one less thing to test and the savings are compounded as you go through the sort.

For example, if the array has 10 elements, then as before in the worst case, we would execute the innermost IF statement 9 * 9 = 81 times. Now, in the worst case, we would execute 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 will make the sort go twice as fast (take half as long to execute).

11.6 Self Test

1 Which Repetitive Statement would you use if you could determine, before you started, how many times your statement(s) would have to be executed?

2 Which Repetitive Statement does not require a BEGIN and END to execute more than one statement repetitively?

3 Which Repetitive Statement(s) test the condition at the beginning of each loop?

4 Rewrite the following WHILE statement as a REPEAT statement. Describe the difference(s) in how it will be executed.

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

Self Test Answers

1 Which Repetitive Statement would you use if you could determine, before you started, how many times your statement(s) would have to be executed?

The FOR statement

2 Which Repetitive Statement does not require a BEGIN and END to execute more than one statement repetitively?

The REPEAT statement

3 Which Repetitive Statement(s) test the condition at the beginning of each loop?

The FOR statement and the WHILE statement

4 Rewrite the following WHILE statement as a REPEAT statement. Describe the difference(s) in how it will be executed.

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

```
REPEAT
  Y := A * X * X + B * X + C;
  X := X - 1;
UNTIL X <= 0;
```

The difference is that if X starts out less than or equal to zero, no statements in the loop will be executed in the case of the WHILE statement, while in the case of the REPEAT statement, the two statements in the loop will be executed once.

In the case that X starts out greater than zero, there is no difference between the way these will be executed.

Chapter 12
Other Statements

We cover two statements, the WITH statement and the CASE statement, in this chapter. The sections will cover examples of these statements. There is no self-test section for this chapter.

12.1 The WITH Statement

The WITH statement is used to make coding with record variables easier. So before we go into the WITH statement, we need to understand what a record variable is.

Record Variables

A record variable is a complex data type (see the chapter on arrays). Like an array variable, a record variable contains multiple values. In an array, all of the values have the same name and type and are distinguished by an element number, or index.

In a record, the various values, also called fields, each have their own name and type. To distinguish the values, a period (.) is used to separate the name of the record variable from the name of the field. Thus, if a record called Customer had a field called Name, then to access the name of the customer, you would enter Customer.Name in your code.

Records represent one row, or record, in a database table. The fields that make up a record are defined using a Table Object. How this is done is covered in the Solution Developer class.

The WITH Statement Syntax

Suppose you had some data in an array that was to be loaded into a record variable. The code to do this might look like this:

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

The WITH statement is used to make this sort of coding easier to do and under many circumstances easier to read. The syntax of the WITH statement is:

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

Within the statement (which can be a compound statement), the record name is no longer needed. Instead, the fields of that record can be used as though they were variables themselves. Thus, the above example could be rewritten like this:

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

Note that the record name is no longer needed. The field name will compile and the computer knows that these fields go with the Customer record. However, if there is a variable name that has the same name as a field, how does C/SIDE know which you are referring to? It uses the field from the record, rather than the variable, but this sort of ambiguous reference should be avoided.

Implied WITH Statements

In many important places that use the WITH statement, you do not even see a WITH statement. These are the implied WITH statements found in the various objects.

For example, in a Table Object, there is an implied WITH statement covering the entire object. Every reference to a field defined in that table has the implied record variable (called "Rec") and a period automatically and invisibly placed in front of it.

In your Solution Developer class, you will learn about other places where there are implied WITH statements.

12.2 CASE Statement

Like the IF statement, a CASE statement is a conditional statement. The IF statement is used when the condition has only two values, either True or False. If the condition is True, the statement following THEN is executed. If the condition if False, the statement following ELSE is executed.

Case Statement Syntax

The CASE statement is used when there are more than two possible values for the condition. The syntax of the CASE statement is as follows:

```
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 is like the constant set used with the IN operator, except without the brackets. Thus, it can contain a single value, multiple values separated by commas, ranges, multiple ranges separated by commas, etc. All of the values in the value sets must have a type compatible with the type of the CASE expression.

Note that the ELSE clause is optional for the CASE statement, thus the brackets.

When the CASE statement is executed, the expression is evaluated first. This expression is sometimes called the selector. Then, in turn, each of the values in each value set is evaluated.

If one of the values in the first value set matches the value of the expression, then the first statement is executed. If one of the values in the second value set matches the value of the expression, then the second statement is executed. This continues until one of the statements is executed.

If the last value set has been reached and no statement has been executed, then the ELSE clause is checked. If not, the second value set is checked. If there is an ELSE clause, then its statement is executed. If there is no ELSE clause, then no statement will be executed for this CASE statement.

Note that only one of the statements will be executed. If there is more than one value set that contains the same value, only the first one of them will be executed.

Simple Use of the CASE Statement

Go into the Object Designer, select codeunit 95100 and click **Design**.

In the OnRun trigger, enter the following statements, removing any code that was there previously.

```
Int2 := 3;

FOR Int1 := 0 TO 8 DO BEGIN
  Color := Int1;

  CASE Color OF
    Color::Orange:
      Int2 := Int1 + Int2;
    Color::Red,Color::Green:
      Int2 := Int2 * 2;
    Color::Yellow:
      BEGIN
        Amt1 := Int1 * 5;
        Int2 := Amt1 DIV 2;
      END;
    Color::Blue,Color::Violet:
      BEGIN
        Amt1 := Int1 * 2 + Amt1;
        Int2 := Amt1 DIV 3;
      END;
    ELSE
      BEGIN
        MESSAGE('Invalid Color');
        Int2 := Int2 + 3;
      END;
  END;

  MESSAGE('The value for %1 is %2',Color,Int2);
END;
```

Before saving this code, try executing this by hand and try to predict what the various messages will say.

If you forget what the various colors were, use **View, C/AL Globals** to display the variable list, select the Color option variable and then press F5 to display the Properties. Make a note of the predicted messages here or on a separate piece of paper.

Now exit, save and **Run** the codeunit.

Did you predict the messages correctly? If not, please review the code and your hand execution to determine what went wrong.

Complex Use of CASE Statement

Now go into the Object Designer, select Form 95100 and click **Design**.

Select the Execute button and access its OnPush trigger code. Modify the code at the beginning (before the comment) so that it looks like this:

```
Result := Quantity * UnitPrice;

CASE TRUE OF
  Quantity = 0:
    EXIT;
  Quantity < 0:
    BEGIN
      TotalCredits := TotalCredits + Result;
      TotalQtyCredited := TotalQtyCredited + Quantity;
      CreditCounter := CreditCounter + 1;
            AvgQtyCredited := TotalQtyCredited DIV
        CreditCounter;
    END;
  Quantity > 0:
    BEGIN
      TotalSales := TotalSales + Result;
      TotalQtySold := TotalQtySold + Quantity;
      SalesCounter := SalesCounter + 1;
      AvgQtySold := TotalQtySold DIV SalesCounter;
    END;
END;

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

Note that we have replaced the various IF statements with one CASE statement.

Look carefully at the way we constructed this CASE statement. Normally in a CASE statement, a variable expression is used for the selector and each value set is a list of one or more constants. In this case, we have used a constant for the selector and each value set is a variable expression. However, the rules for execution are still the same. The selector expression is evaluated and then each of the values in each value set is evaluated. The first value set where both values match is the one that is executed.

Exit, save and **Run** the Form.

Verify that the Execute button works as before.

Note that there is one difference. What is it?

How could you have changed the code so that it executes identically to the previous code?

Chapter 13
Calling Built-in Functions

Functions and Parameters are reviewed in this chapter. This chapter also covers several Built-in Functions that C/AL provide.

13.1 Functions and Parameters

Fully understanding functions and parameters and how they are used is very important. So we first will review these terms and others that are closely related.

Functions

A function is a named portion of a program (sometimes called a "sub-program" or a "subroutine").

When the function's name, also called an identifier, is used, the current program is suspended while the trigger code for the specified function is executed. Using the Identifier in this way "calls" the function. When the trigger code in the called function is completed, the function "returns" to where it was called. Depending upon how the function is called will determine what happens when it returns.

A function can be used as an expression. For example, here is a function named CalculatePrice used in the following expression:

```
TotalCost := Quantity * CalculatePrice;
```

The CalculatePrice function will need to return a value, which is used in evaluating the expression. This return value is then multiplied by the variable Quantity and that result is assigned to the variable TotalCost.

A function can also be called using a function call statement. This statement simply calls the function and does not receive a return value. Here's an example of calling a function named "RunFunction" in this manner:

```
IF Quantity > 5 THEN
  RunFunction;
```

The RunFunction function does not return any data back to the calling function.

Built-in Functions

A built-in function is a function provided within C/SIDE without you having to write it. You will not be able to see its trigger code, since that too is built into C/SIDE.

Parameter

A parameter is one or more variables or expressions that are sent to the function through the function call. The parameter provides the function information and, if needed, the function can modify that information. Most functions will have parameters and if so, the function identifier will have a set of parenthesis, that follow the function identifier. Within these parentheses will be one or more parameters. If there is more than one parameter, the parameters will be separated by commas.

Pass By Value

Sometimes a parameter is passed to a function strictly to give the function information. In that case, the parameter is said to be passed (or called) by value. In this case, the parameter knows just the value of the variable or expression that was used for the parameter. Since it is just a value, any change that the function does to this parameter does not affect any variables in the calling trigger.

Pass By Reference

Other times, a parameter is passed to the function and the function modifies that parameter. In this case, the parameter is said to be passed (or called) by reference or by name. The parameter actually knows the variable's location in the computer memory that it represents. The parameter passes the computer memory location to the new function. Any changes that the function does to this type of parameter is permanent and does affect variables in the calling trigger.

These concepts are covered more thoroughly in the next chapter. However, there is one important thing to note at this point. If a parameter is passed by value, then you can use any expression for that parameter. However, if a parameter is passed by reference, you must use a variable for that parameter, so that its value can be changed. A variable has a location in memory, whereas an expression or a constant does not.

13.2 Using Built-in Functions

We have already covered several built-in functions in previous chapters. Take a couple of minutes to review these:

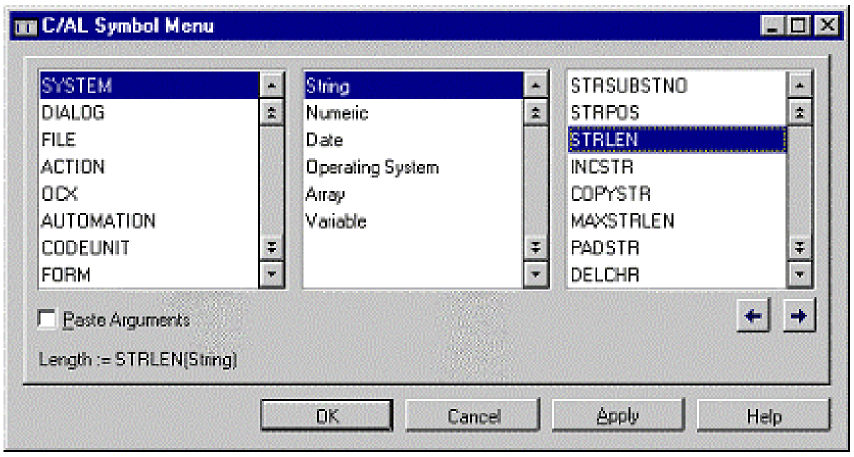
- `MESSAGE` (displays a message on the screen) see chapter 3
- `MAXSTRLEN` (returns defined length of text variable) see chapter 5
- `COPYSTR` (returns part of a string) see chapter 5
- `CLEAR` (clears the passed in variable) see chapter 10
- `ARRAYLEN` (returns number of elements in an array) see chapter 10

Note that some of them, like MESSAGE and CLEAR, have no return value, so they can only be called using a function call statement. Others, like COPYSTR and MAXSTRLEN, return a value, so they can be used in an expression. Also, note that CLEAR changes the passed in parameter, so it is an example of pass by reference, while the others are examples of pass by value.

The C/AL Symbol Menu

Using the Object Designer, go into codeunit 95100 again.

Now select **View, C/AL Symbol Menu** from the Menu Bar, or press the F5 key. You can also press the Symbol button on the Tool Bar. The following window should appear:



Length := STRLEN(String)

If this does not immediately appear, scroll the left-hand panel until the function groupings appear in it, as shown above.

Highlight the DIALOG choice to see some of the built-in functions that display on the screen. In the right-hand panel, highlight the MESSAGE function. Note that at the bottom of the frame (above the command buttons), a line appears that looks like this:

```
MESSAGE(String [, Value1] …)
```

This is to tell you the syntax of the function. Note that there is no return value and that there is one required parameter (String) and multiple optional parameters. The square brackets indicate an optional parameter and the ellipsis indicates multiples.

Now, highlight the ERROR function (just above MESSAGE). As you can see, its syntax looks like this:

```
ERROR(String [, Value1] …)
```

The syntax of the ERROR function is identical to that of the MESSAGE function (except for the function identifier and the functionality.) When an ERROR function is called in a function call statement, the processing stops with an error condition and the message is displayed in a similar manner as the MESSAGE function.

Now, in the left panel, highlight SYSTEM. Note that in the middle panel, there is now a list of various function groupings, like string functions, numeric (mathematical) functions, and so on. As you highlight each of these choices in the middle panel, the functions listed in the right-hand panel change. As an example, highlight "Date" in the middle panel and then, in the right-hand panel, highlight DATE2DMY. The syntax of this function now appears at the bottom of the window:

```
Number := DATE2DMY(Date, What)
```

The "Number :=" listed before the function identifier indicates that this function has a return value and that this return value is a number. The first parameter, "Date" is an expression of type date. But what could "What" (the second parameter) be? To find out, now that you have highlighted the DATE2DMY function, press the F1 key.

The help screen displays for this function and you can read a complete description of what it does and what it returns. There, you will find out that "What" is an integer expression that should resolve to one of three values. If it is a 1, then this function returns the day of the month. If it is a 2, then this function returns the month (from 1 to 12). If it is a 3, then this function returns the year (the full 4 digits).

Using the DATE2DMY Function

Here is an example of this function, which extracts the month from a date type variable and displays it as human readable text. You can try it in your Workbook Exercises codeunit to see how it works.

```
"When Was It" := TODAY;
CASE DATE2DMY("When Was It",2) OF
  1:Description := 'January';
  2:Description := 'February';
  3:Description := 'March';
  4:Description := 'April';
  5:Description := 'May';
  6:Description := 'June';
  7:Description := 'July';
  8:Description := 'August';
  9:Description := 'September';
  10:Description := 'October';
  11:Description := 'November';
  12:Description := 'December';
END;
MESSAGE('%1 is in %2',"When Was It",Description);
```

Note that the first line in that code uses another function, called TODAY. This function has no parameters. It always returns the current date from your computer's operating system, also known as the "system date".

13.3 Self-Test

The questions on this self-test relate to the following code. You should hand execute this code in order to find the answers, using the Symbol Menu and the Help system to determine what the various functions do. Note that all of these functions can be found in the String subsection of the SYSTEM section. Do not actually run this code in your test codeunit until you reach question 8.

// UserInput is a Text variable of length 100. The other variables are Integers.

```
UserInput := 'The colors are red, orange, yellow,
green, blue and violet.';
Count := 0;

REPEAT
  Comma := STRPOS(UserInput,','); //Q1
  IF Comma > 0 THEN BEGIN //Q2
    Count := Count + 1;
    UserInput := DELSTR(UserInput,Comma,1); //Q3
    UserInput := COPYSTR(INSSTR(UserInput,
      ' and',Comma),1,MAXSTRLEN(UserInput));
  END;
UNTIL Comma <= 0;

// Display Results
MESSAGE('The sentence is "%1". Number of commas is
%2.',UserInput,Count);
```

1. What is the value of Comma after the statement labeled Q1 is executed the first time?

2. Is the value of Comma ever less than 0 in this code (see statement labeled Q2)?

3. What is the value of UserInput after the statement labeled Q3 is executed the first time?

4. What is the value of Count when the comment line is reached the first time?

5. Suppose that UserInput was redefined so that it was a Text of length 60. Now, what is its value when the MESSAGE function after the comment line

is reached?

Note that the end of the sentence can be lost under the above circumstances. Let's suppose that you modify the UNTIL clause of the REPEAT statement so that it looked like this:

```
UNTIL (Comma <= 0) OR (STRLEN(UserInput) + 3 >
MAXSTRLEN(UserInput));
```

6 Now, what is the value of UserInput when the MESSAGE function is reached?

7 Although better, this was not the desired result. Why didn't it work?

8 Go ahead and run the above code in your test codeunit. Don't forget that you will have to define some new variables. Now, make as simple change as you can to address the problem you uncovered in question 7. Write your change here:

Self Test Answers

1 What is the value of Comma after the statement labeled Q1 is executed the first time?

19

2 Is the value of Comma ever less than 0 in this code (see statement labeled Q2)?

No

3 What is the value of UserInput after the statement labeled Q3 is executed the first time?

The colors are red, orange, yellow, green, blue and violet.

4 What is the value of Count when the comment line is reached the first time?

4

5 Suppose that UserInput was redefined so that it was a Text of length 60. Now, what is its value when the MESSAGE function after the comment line is reached?

The colors are red and orange and yellow and green and blue

6 Now, what is the value of UserInput when the MESSAGE function is reached?

The colors are red and orange, yellow, green, blue and violet

7 Although better, this was not the desired result. Why didn't it work?

Because a REPEAT loop is always executed at least once and the test was done after the variable had already been changed.

8 Go ahead and run the above code in your test codeunit. Don't forget that you will have to define some new variables. Now, make as simple change as you can to address the problem you uncovered in question 7. Write your change here:

Here is one possible answer:

```
UserInput := 'The colors are red, orange, yellow,
                   green, blue and violet.';
Count := 0;

WHILE (STRPOS(UserInput,',') <> 0) AND
      (STRLEN(UserInput) + 3 <= MAXSTRLEN(UserInput))
        DO BEGIN
  Comma := STRPOS(UserInput,',');
  Count := Count + 1;
  UserInput := DELSTR(UserInput,Comma,1);
  UserInput := INSSTR(UserInput,' and',Comma);
END;

MESSAGE('The sentence is "%1". Number of commas is
%2.',UserInput,Count);
```

Note that the REPEAT loop was changed into a WHILE loop. In order to do this, the condition had to be logically negated, since a REPEAT loop continues as long as the condition is FALSE, while a WHILE loop continues as long as the condition is TRUE. To logically negate any logical expression, you could put a NOT in front of it and enclose the entire expression in parentheses. Or you could do as we did above: Change every relational operator to its opposite (“=" to "<>" and vice versa, ">" to "<=" and vice versa and "<" to ">=" and vice versa), plus change every AND to an OR and every OR to an AND.

Chapter 14
Creating Your Own Functions

This chapter covers why one should create functions and how to create them. The EXIT statement is also reviewed.

14.1 Formal and Actual Parameters

Formal Parameter

This is a parameter as defined in the function definition. In the previous chapter's exercise, we used the DELSTR function. When you look up the DELSTR function in the Symbol Menu, it listed the syntax as:

```
NewString := DELSTR(String, Position [, Length])
```

The words that appear in parentheses are the formal parameters. If you were able to see the trigger code for a built-in function, these formal parameters could be used as variables in that trigger code.

Actual Parameter

The actual parameter is what you use when you call the function. When we called the DELSTR function in the above-mentioned exercise, we used the following line:

```
UserInput := DELSTR(UserInput,Comma,1);
```

The constant and variables that appear in the parentheses are the actual parameters.

Note that there is a one-to-one correspondence between the actual parameters and the formal parameters. The actual parameter "UserInput" becomes the formal parameter "String", the actual parameter "Comma" becomes the formal parameter "Position" and the actual parameter "1" becomes the formal parameter "Length".

Since we are using pass by value for all three parameters, what we are actually passing to the function are the values of the three actual parameters. Thus, the actual parameters are not changed when the formal parameters are changed inside the function.

If we had used pass by reference, we would be passing the variable references (memory addresses) to the function. Then the actual parameters would be changed if the corresponding formal parameters were changed inside the function. Naturally, since constants cannot be changed, we would not have been able to use "1" as an actual parameter if Length were being passed by reference; it would have generated a syntax error.

14.2 Local Functions and Variables

Local Function

A local function is a function that can only be called in the object in which it is defined. Any function that has not been defined as a local function can be called from other objects as well as the object in which it is defined.

Local Variable

A local variable is a variable whose scope is limited to a single function. This means that in this function's trigger code, a local variable can be used like any other variable. However, throughout the rest of the object, this variable cannot be accessed at all. If the name of a local variable is referred to outside of the function in which it is defined, a syntax error will result. The formal parameters of a function are also treated as local variables in that function.

14.3 Why Create Functions?

We have been able to create programs that are quite useful without using functions. Why should we create them now? It is true that you can program without creating functions. Also, for almost anything you might need to do in C/SIDE, you will not have to create functions in order to create the functionality required. However, there are some very good reasons why programmers create functions. Here are a few:

- To organize your program. A function is to your code what the headings are to a well-organized written document. They make your program easier to follow, not only for yourself, but also to other programmers who may need to modify your code later.

- To simplify your tasks. When you are designing your program, you can break a complex problem into multiple smaller tasks. Each of these tasks can become a function in your program and the whole program can be put together from these smaller tasks. Should a function turn out to be too complex, you can do the same thing again: break it apart into smaller tasks and create a new function for each task.

- To reduce your work by re-using code. If you find that you are doing the exact same thing, or very similar things, in two separate parts of your program, consider creating a function to do that task. Then, rather than writing the same or similar code in two or three places, you can write it in one place and call it from other places.

- To reduce the chance of errors. A function can be tested thoroughly by itself, using special test programs to test all possibilities. Thus, when it is used in another place, it is a known quantity; when you are searching for errors, you can reduce your search. Similarly, should an error be found in a function, it can be fixed in one place and it will automatically be fixed in all the places that called that function. If you had to fix it in each place, then you might forget one, or you might even add another bug.

- To make modifications easier. If you do need to modify the way your program works and you have similar code in many places, you will have to make your modifications to each of those places. If you have put that common code in a function, you can make your modification to one place and every place that uses that function will be updated as well. This reduces your work and reduces the chance of introducing more errors.

- To localize data. When a function performs a task, it can have its own local data that cannot be tampered with by other functions in the same object. By using its own local data, it will not tamper with data owned by those other functions. If you have global variables used by many tasks throughout your program, there is a good chance that this data may become corrupted.

- To reduce the size of your objects. Although a minor consideration, it can be worthwhile, especially when you are trying to locate something in your code.

There are many good reasons to create functions. In fact, when you are designing code for an object, the first thing you should do is figure out your major tasks and create (define) a function for each one. Then, as you discover yourself doing similar things in different places, always consider adding another function to handle that task.

Another reason to create functions, which is specific to C/SIDE, is that functions are one of the main ways to communicate between objects. Remember that variables cannot be shared across objects, but functions can be. This will be discussed in more detail in your Solution Developer class.

14. 4 The EXIT Statement

As you recall, the EXIT statement is used to stop the execution of a trigger. It can be used in this manner for function Triggers as well. However, in functions, the EXIT statement has an additional use.

When a function call is used in an expression, the function is required to return a value. When you write your own function that has a return value, you will signal to the system to return this value by using the EXIT statement. Below is the syntax:

```
EXIT(<expression>);
```

For example, let's create a function named "Square" which is used to square a value. The following expression...

```
Answer := 4 + Square(5);
```

...would result in Answer being assigned the value 29. The Square function's trigger code could be written like this if the formal parameter is called "Param"

```
EXIT(Param * Param);
```

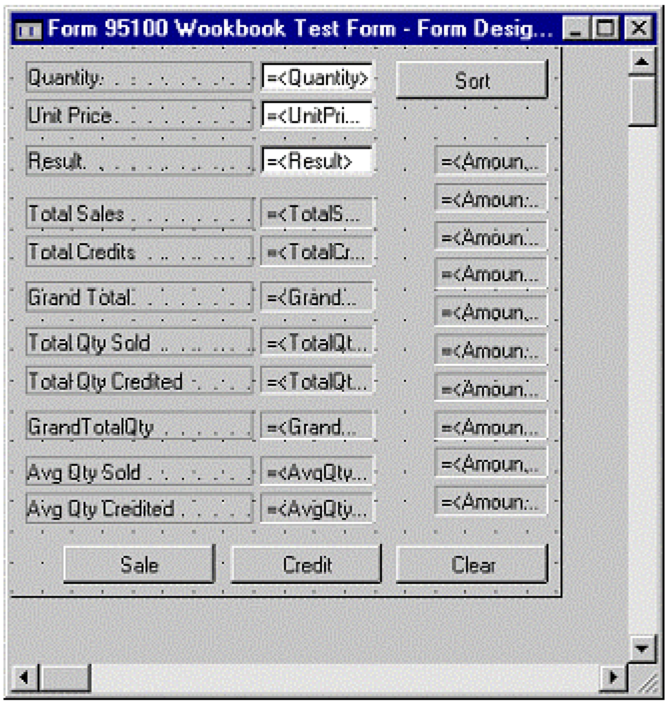
The EXIT statement's parameter was the expression that squared the parameter and that is what the function will return to its caller.

14.5 Creating Functions

Go into the Object Designer, select Form 95100 and click **Design**. Move the Sort Command Button control by dragging it to the upper right-hand corner of the form.

Select the Execute Command Button control. Press the Copy button on the Tool Bar (or press Ctrl+C on the keyboard). Click any place on the form that does not have a control on it. Now, press the Paste button on the Tool Bar (or press Ctrl+V on the keyboard). A copy of the Execute button will be placed in the upper left corner of the form. Now drag this copy next to the original Execute button.

Go into the Properties for the left-hand button and change the Caption property to "Sale". Now, select the original Execute button and change the Caption property to "Credit". At this point, your form should look like this:



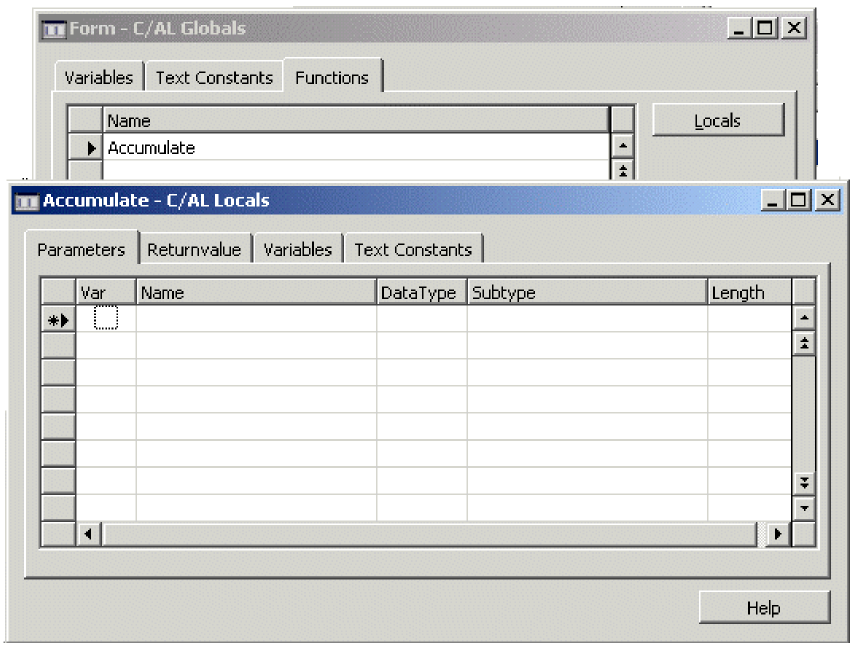
Defining a Function

Now, go into the Globals list by selecting **View, C/AL Globals** on the Menu Bar. Note that there are three tabs on this form and, up until now, we have always used the first tab. This time, click the 3rd tab, **Functions**. This is where we go to define a Function. The cursor should now be in the Name column. Enter the name of our first function, which will be:

```
Accumulate
```

While on the line that now says Accumulate, display the Properties form. Note that one of the Properties is called Local. Since we do not want this function to be called from any object outside of this form, set this property to Yes. This will make our new function a Local Function. Close the Properties window.

Back on the Globals form, press the Locals button. Another form comes up, called Locals. This form is used to complete the definition of your new Accumulate function, so it should say Accumulate in its title bar, as shown here:



By the three tabs at the top of this form, it is apparent that you can set up the formal Parameters for the function, the Return Value and Local Variables you may need. On this function, we just need a single parameter, so enter Qty as the Name and Integer as the data type. The column labeled Var is used if you want this parameter to be passed by reference. A parameter passed by reference must be a variable (not an expression) and that is why this column is labeled Var, which is short for Variable. In this case, we will not check this column.

Close this form so you can return to the Globals form, still on the Functions tab. We have completed the definition of the Accumulate function.

Defining Additional Functions

Now, add another line to the Function List, this time named:

```
Extend
```

Set its Local property to Yes, as we did above for the Accumulate function. Now press the Locals button and create two parameters. The first should be an Integer parameter named Qty and the second should be a Decimal parameter named Unit.

Now, select the middle tab on the Locals form, "Returnvalue". Click on the down-arrow button in the Return Type field to bring down the list of possible return types. Select Decimal as the Return Type of this function.

Close the Locals form. This completes the definition of the Extend function.

Add a third line to the Function List, this time named

AddToArray

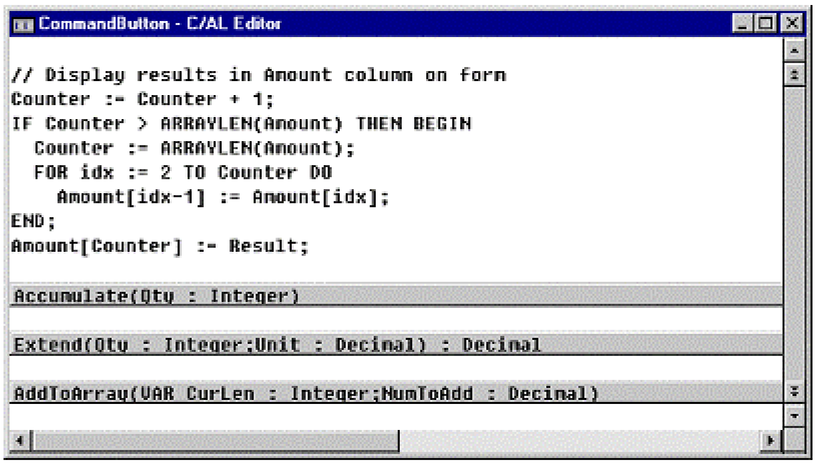
Set its Local property to Yes and then press the Locals button. Add an Integer parameter called CurLen. This time, click in the Var column, so that a check mark appears to the left of CurLen. This sets CurLen to a Var parameter, which means it will be passed by reference rather than by value.

Add a second parameter, called NumToAdd, with a type of Decimal. This one will not be a Var parameter, so do not check the Var column. It will be passed by value.

This time, select the right tab on the Locals form, the one named, Variables. Add an Integer variable named idx. Then, close the Locals Form and close the Globals form.

Adding Code to your Functions

Click on the Sale Command Button and then on the Code button (or press the F9 key). Scroll down and you should see your three new functions, looking like this:



```
// Display results in Amount column on form
Counter := Counter + 1;
IF Counter > ARRAYLEN(Amount) THEN BEGIN
  Counter := ARRAYLEN(Amount);
  FOR idx := 2 TO Counter DO
    Amount[idx-1] := Amount[idx];
END;
Amount[Counter] := Result;

Accumulate(Qty : Integer)

Extend(Qty : Integer;Unit : Decimal) : Decimal

AddToArray(VAR CurLen : Integer;NumToAdd : Decimal)
```

If these three Function Trigger lines do not look like the example, go back to View, Globals and correct the function definitions you entered above.

Now, fill in the trigger code for each of these three functions, as shown below. Note that all of this code is very similar to the code that is in the OnPush trigger of the Sale button. If you feel comfortable with it, you can copy the code from there and paste it into these functions. Then modify the code to match the following:

```
Accumulate:

Result := Extend(Qty,UnitPrice);
IF Qty < 0 THEN BEGIN
  TotalCredits := TotalCredits + Result;
  TotalQtyCredited := TotalQtyCredited + Qty;
  CreditCounter := CreditCounter + 1;
  AvgQtyCredited := TotalQtyCredited DIV CreditCounter;
END ELSE BEGIN
  TotalSales := TotalSales + Result;
  TotalQtySold := TotalQtySold + Qty;
  SalesCounter := SalesCounter + 1;
  AvgQtySold := TotalQtySold DIV SalesCounter;
END;
GrandTotal := GrandTotal + Result;
GrandTotalQty := GrandTotalQty + Qty;
```

```
Extend:

EXIT(Qty * Unit);
```

```
AddToArray:

CurLen := CurLen + 1;
IF CurLen > ARRAYLEN(Amount) THEN BEGIN
  CurLen := ARRAYLEN(Amount);
  FOR idx := 2 TO CurLen DO
    Amount[idx-1] := Amount[idx];
END;
Amount[CurLen] := NumToAdd;
```

Calling your Functions

Now, scroll up to the OnPush code of the Sale Command Button and modify it so it looks like this:

```
IF Quantity = 0 THEN
 EXIT;
Accumulate(Quantity);
AddToArray(Counter,Extend(Quantity,UnitPrice));
```

Close the code form and then click on the Credit Command Button. Display the code form again and modify the OnPush trigger code of the Credit Command Button so it looks like this:

```
IF Quantity = 0 THEN
 EXIT;
Accumulate(-Quantity);
AddToArray(Counter,Extend(-Quantity,UnitPrice));
```

Note that both routines are small and they are very similar. The only difference is that the Credit button turns the Quantity negative before calling the functions. This means that the user will not enter negative quantities any more. Instead, they will just enter quantities and then press either the Sale or the Credit button, depending on what kind of transaction it is.

Close and save this Form Object, then run it and try it out.

14.6 Self-Test

Go into the Object Designer and find codeunit 1 (named "ApplicationManagement"). Open it and answer the following questions by looking at this object. Remember: If a question is about the Function Definition, it would be best to look at the Globals form (the Functions tab) and the Locals form. If a question is about the code, it would be best to look at the trigger code.

1 How many Parameters does the LoginStart function have?

2 What type of value (if any) is returned by the ReadRounding function?

3 In the ReadSymbol function, is the first parameter (Token) passed by value or passed by reference?

4 In the ReadSymbol function, is the second parameter (Text) passed by value or passed by reference?

5 In the AutoFormatTranslate function, how many local variables are defined?

6 Look at the code for the AutoFormatTranslate function. How many lines in that trigger code could return the value of this function?

7 Look at the code and the definition of the ApplicationLanguage function. Is there anything in this function that affects any data outside this function (not counting the return value)? If so, what?

8 Look at the code and the definition of the ReadCharacter function. Is there anything in this function that affects any data outside this function (not counting the return value)? If so, what?

Chapter 15
C/AL Functions

This chapter lists functions that C/AL provides for strings, numbers, and dates, among others. The syntax for each function is shown and for the more commonly used functions, some examples.

This section will cover the following:

15.1 User Communication Functions

The following functions give the user feedback and/or an opportunity for them to input information into the program –

MESSAGE

MESSAGE (String [, Value1, ...])

The message function is run non-modally, meaning the remaining code in the process is run before the message. It displays a message to the user, but only after the process is complete. The window remains open until the user selects the OK button (or presses the Enter key)

This function is often used for debugging purposes such as displaying values of variables. However, if an error is occurring and since it is run non-modally, the process will stop and the message will not display.

When working with strings, the plus character ("+") is used to concatenate text and the backslash character ("\") will start a new line. The "%" and "#" symbols may be used as variable placeholders. The percent is used for free format and the pound symbol is for fixed formats.

Example:

Value1 := 12345.678;

Value2 := 987.65

MESSAGE('The Format of Value1 is %1 \' + ' The Fixed Format of Value2 is #2#########', Value1, Value2);

The message window shows:

The Free Format of Value1 is 12,345.678

The Fixed Format of Value2 is 987.65

CONFIRM

CONFIRM (String [, Default] [, Value1, ...])

This function is similar to the MESSAGE function, but allows the user to answer a question by selecting a Yes or No button. Then, the function returns a Boolean value (True or False), corresponding to the user's selection. Since this is run modally, the system waits for the user's response.

CONFIRM is often used to confirm that the user wants to continue with a given process. Navision Financials uses the CONFIRM functions prior to posting records. The user is given an opportunity to stop the posting process or continue with it.

The Default value is False, as are normal Boolean data types, therefore, the No button is the default button and is active. This means if the user just presses "Enter", the function will return a False. Normally, if the user will select the Yes button, then by setting the Default parameter to "TRUE" will set the Yes button to the default.

Example:

```
IF NOT CONFIRM('Do You  want to post the Journal Lines?') THEN

  EXIT()

// Otherwise run the posting routine.
```

The Confirm function is limited as to the options (Yes, No) that it allows. If other options are needed, there is another function that may be more useful, the STRMENU dialog function.

STRMENU

STRMENU (OptionString [, DefaultNumber])

This is used to create and display a form with an option group, returning the value (integer) of the user's selection. If no default is provided, the default is the first element in the option string, with a value of 1. A zero value is returned if ESC is pressed, indicating no choice by the user.

Example:

```
MESSAGE('Your selection returns the value of %1', STRMENU('Yes, No,
N/A'));
```

ERROR

ERROR(String [, Value1, ...])

This raises an error condition and leaves the current process, canceling the entire process (not just the function). So, if the code is in a transaction, the transaction is stopped and all uncommitted data is rolled back.

It returns an Error message (String) to the user, which should inform them why further processing was not allowed.

Example:

```
IF Number <= 0 THEN BEGIN

  ERROR('Number must be positive.  Current value: %1.', Number);

  MESSAGE('This Message will never be displayed');

END;
```

15.2 String Functions

These functions allow the programmer to manipulate strings. For more help, use the Navision Financials online documentation.

STRPOS

Position := STRPOS(String, SubString)

The STRPOS function is used to search for the first occurrence of a sub-string within a string, returning the position of the first character in the sub-string. If SubString is not found within String, the function returns a zero. As with all string functions, STRPOS is case sensitive.

COPYSTR

NewString := COPYSTR(String, Position [, Length])

The COPYSTR function is used to copy a sub-string of any length from a specific position in a string to a new string. If no length is provided, the result will include all characters from Position to the end of the string.

PADSTR

NewString := PADSTR(String, Length [, FillCharacter])

This is used to change the length of a string to a length you define. If the parameter 'Length' is smaller than the actual length of the string, then the string is truncated. Otherwise it adds filler characters to the end of the string. If the parameter FillCharacter is not provided, then blanks will be added.

There are also two other functions that are similar to PADSTR. One is **DELSTR**, which is used to delete a sub-string from inside a string. There is also **INSSTR**, which inserts a sub-string into a string at a specified position. The Syntax for these functions is as follows:

NewString := DELSTR(String, Position [, Length])

NewString := INSSTR(String, SubString, Position)

STRLEN

Length := STRLEN(String)

This function returns an integer, which is the length of the string in the parameter.

MAXSTRLEN

MaxLength := MAXSTLEN(String)

Similar to STRLEN, it also returns an integer, but that integer represents the maximum (defined) length for that string variable.

LOWERCASE and UPPERCASE

NewString := LOWERCASE(String),

NewString := UPPERCASE(String)

Use these functions to convert a string to all lower-case or upper-case, respectively.

CONVERTSTR

NewString := CONVERTSTR(String, FromCharacters, ToCharacters)

This function converts the characters in a string based on the characters in the strings FromCharacter and ToCharacters, which serve as conversion tables.

DELCHR

NewString := DELCHR(String [, Where] [, Which])

This is used to delete one or more characters in a string. It can delete characters from either end of the string or all instances throughout the string. Note that this can be used to trim strings of blanks:

```
DELCHR(StringVar, '<>', ' ')
```

INCSTR

NewString := INCRSTR(String)

Use this function to increase a positive number or decrease a negative number inside a string by one (1). If there is more than one number in a string, it will change the first number.

SELECTSTR

NewString := SELECTSTR(Number, CommaString)

This function retrieves a substring from a comma-separated string. The substrings are numbered starting with 1.

STRCHECKSUM

```
CheckNumber := STRCHECKSUM(String [,WeightString] [, Modulus])
```

This is used for a variety of different applications such as barcodes. It calculates a checksum for a string containing a number.

15.3 System Functions

These functions do not require any parameters since they return information that is stored in the system.

USERID

Name := USERID

This function returns the ID of the current user.

COMPANYNAME

Name := COMPANYNAME

This function returns the current company that the program is using.

TODAY and TIME

DateVar := TODAY, TimeVar := TIME

These functions return the operating system's date and time.

WORKDATE

[WorkDate] := WORKDATE([NewDate])

This function returns the current workdate or may be used to reset the workdate. Although the developer can change the workdate, note that in standard Navision applications, ONLY the USER should change the Workdate, it should not be changed in code.

15.4 Date Functions

These functions provide specific information about a given date.

DATE2DMY

IntegerVar := DATE2DMY(Date, Integer)

This function returns information about the parameter 'Date'. Set 'Integer' to '1' to return the day of the month (1-31), set Integer to '2' to return the month (1-12), or to '3' to return the year (0000-9999).

DATE2DWY

IntegerVar := DATE2DWY(Date, Integer)

This function also returns information about the parameter 'Date'. Set 'Integer' to '1' to return the day of the week (1-7, remember that day 1 is Monday), set Integer to '2' to return the Week number (1-53), or to '3' to return the year (0000-9999).

CALCDATE

CALCDATE(DateExpression [, Date])

CALCDATE is a very powerful function. It can calculate a new date based on a date expression and a reference date. Calculating dates based on document dates, posting dates, invoicing dates, and so on, is common throughout Navision Financials.

Here are some examples of a DateExpression:

'CQ + 1M – 10D':	Last Day of Current Quarter + 1 Month – 10 Days
'-WD2':	Last 2nd Day of the Week, (last Tuesday)
'CM + 30D':	Last Day of Current Month + 30 Days

Calcdate('CM +15D', 030502D) returns 04/15/02.

Calcdate starts with the March 5, 2002 reference date and goes to the last day of that current month (CM). This brings us to 03/31/02. Then it adds 15 days, which is April 15, 2002.

The date expression can be any length; it must have at least one SubExpression. The system interprets the string from left to right, one SubExpression at a time. The SubExpression is either a positive or negative Term value.

The Term value may be a Number–Unit combination, a Unit–Number combination or a Prefix–Unit combination. The Number is a positive integer. The Unit may be one of the following:

D	(Date)
WD	(WeekDay)
W	(Week)
M	(Month)
Q	(Quarter)
P	(Period)
Y	(Year)

There is one Prefix: C, which stands for the Closing date of that period. The closing date for a quarter would represent 12:59 PM of the last day of the last month in the quarter.

Every date, from 01/01/0000 to 12/31/9999 has a corresponding normal date and closing date, used for closing journal entries. To the system, a closing date represents 12:59 PM of that date. Closing dates are sorted immediately after the corresponding normal date, but before the next normal date. The letter “D” indicates a normal date, and “C” indicates a closing date. These functions determine which type of date is being used:

NORMALDATE

```
ReturnDate := NORMALDATE(Date)
```

A very common place to see this function is in the posting routines. For example posting dates or invoice dates must be Normal dates. The following code ensures that only Normal dates are posted.

Example:

```
IF "Posting Date" <> NORMALDATE("Posting Date") THEN

  ERROR(‘Posting Date cannot be a closing date');
```

CLOSINGDATE

```
ReturnDate := CLOSINGDATE(Date)
```

This function is the opposite of the above function and can be used to return the corresponding Closing date.

15.5 Number Functions

Most of these are rarely used, with the exception of the Round function. That function is used quite often when dealing with currency and tax.

ABS

NewNumber := ABS(Number)

This function returns the absolute value of the number.

POWER

NewNumber := POWER(Number, Power)

Raises the Number parameter to the Power parameter. This can also calculate roots. For example, to calculate the square root of 16... POWER(16, .5) returns 4.

ROUND

NewNumber := ROUND(Number [,Precision] [, Direction])

This function returns a rounded number. The Precision parameter determines the precision used when rounding off. The default value is .01 for the US version, however settings in the G/L setups can affect this. The Direction parameter details how to round:

=	Rounds to the nearest value (default)
>	Rounds up
<	Rounds Down.

RANDOMIZE

RANDOMIZE ([Seed])

This function generates a set of random numbers. Seed is an optional integer value that is used to create a unique set of numbers. Using the same seed number results in the same set. If the parameter is omitted, the current system time (total number of milliseconds since midnight) is used.

RANDOM

Number := RANDOM(MaxNumber)

Using the random number set from the RANDOMIZE function, this returns a pseudo-random number between 1 and MaxNumber. Until RANDOMIZE is called again, RANDOM chooses a number from the same set of numbers.

15.6 Array Functions

Arrays are not frequently used in Navision, however they are often used in reports for address and label information.

ARRAYLEN

Length := ARRAYLEN(Array [, Dimension])

Returns the total number of elements in an array or the number of elements in a specific dimension. A 3-Dimensional array would have valid dimensions of 1, 2, and 3. If, in this case no dimension were given, then the return value would represent the number of elements in the whole array, not just a dimension.

COMPRESSARRAY

[Count :=] COMPRESSARRAY(StringArray)

This function is only useful for text or code arrays. It moves all the non-empty strings of the array to the beginning of the array. The same number of elements are in the array, however, the empty entries and those that contain only blanks appear at the end of the array.

If a return value is captured, it represents the number of non-emtpy strings the system compressed.

A good example of using this function would be when printing names and addresses. This function would be useful to remove blank lines in account statements or from multi-line addresses.

COPYARRAY

COPYARRAY(NewArray, Array, Position [, Length])

This copies one or more elements in an array to a new array. Position is the position of the first array element to copy, while the optional Length is the number of array elements to copy. If Length is not included, then all array elements from Position to the last element are copied. This is only used for 1 dimensional arrays. Repeat this function to copy more dimensions.

15.7 Other Important Functions

These are important functions that are used throughout Navision –

EXIT

EXIT (Value)

This command leaves the current function or trigger immediately. If there is a parent function that called this current function or trigger, then the value is returned to the calling function. This does NOT cause an error condition or rollback any data.

Example:

```
Function AddTen (Number : Integer) : Integer

BEGIN

  Exit(Number + 10);

  Message ('This line is never read.')

END;
```

CLEAR

CLEAR(Variable)

This clears the value of a single variable (or all elements of an array) to the following:

Variable Type	Clearing Result
Number	0 (zero)
String	empty string
Date	0D (undefined date)
Time	0T (undefined time)
Boolean	FALSE

CLEARALL

CLEARALL

This parameters-less function clears all internal variables in the current object and in any called objects such as reports, units of code and so on that contain C/AL code. It works by calling CLEAR repeatedly for each variable.

When an object is called repeatedly within the same process, the system retains all values for variables and filters in memory between calls. Using CLEARALL will clear all of this

EVALUATE

[Ok :=] EVALUATE (Variable, String)

Use this function to convert a string expression into another appropriate data type. The result is assigned to the Variable parameter. This function can convert string expressions into Decimal, Integer, Date, Time, Boolean, Option, Text and Code data types.

FORMAT

String := FORMAT(Value [, Length] [, FormatNumber | FormatString])

This is a very powerful function that will take some time to fully master. At the very basic level, it can convert any type of variable to a string variable. The following changes a decimal variable to a string:

TextVar := Format(DecimalVar);

The parameter **Length** (integer) can ensure that if the value is larger than the maximum length that the String allows, then a run-time error does not occur.

If length = 0, then the system will return the entire value (default).

If length > 0, then String will be exactly Length characters. If Value is less than Length characters, the system inserts either leading or trailing spaces, depending on the format you select. If Value exceeds Length characters, the system truncates String accordingly.

If length < 0, then String will have the maximum length of Length characters. If Value is less than Length characters, the length of String will equal the length of Value. If Value exceeds Length characters, the system truncates String accordingly.

Only one of the next two parameters is used at any time. **FormatNumber** determines the format the system will use. The basic options are:

0	Standard Display Format (the default for all data types)

1	Standard Display Format 2 (edit)
2	C/AL Code Constant Format

Other options are available depending upon the datatype. See the online help for more information.

FormatString is more powerful; it is a literal string that defines a format as in the Format property. This property describes the various predefined formats in detail, and also how to create customized formats.

Examples

MESSAGE('The formatted value: %1', FORMAT(-123456.78, 15, 0));

The message window shows – The formatted value: 123,456.78

MESSAGE('The formatted value: %1', FORMAT(-123456.78, 5, 3));
// Changed length to 5

The message window shows – The formatted value: 123,4

MESSAGE('Today is %1', FORMAT(TODAY,0,'<Month Text> <Day>.');

The message window shows – Today is April 15.

A common request is to show negative numbers as strings with parenthesis instead of minus signs. If this is part of a vertical list of numbers, one also might want to add a trailing space at the end of the positive numbers (so that negative amounts, with their added parenthesis, align with the positive amounts). Here is a way to do it using the FORMAT function:

```
IF DecimalVar < 0 THEN

  StringVar := FORMAT(DecimalVar, 0, '(<Integer Thousand><decimals>)')

ELSE

  StringVar := FORMAT (DecimalVar, 0, '<Integer Thousand><decimals> ')
```

Note, the '<' and '>' signs are part of the FormatString parameter. For the negative values, note the parenthesis inside the single quotes and for the positive values, note the space before the last single quote for alignment.