

4GL Training Guide



**4GL Training Guide
Version 6 – March 2012
Revision number 1.03**

Querix 4GL

4GL Training Guide

Part Number: 019-006-103-012

Svitlana Ostnek, Elena Krivtsova, Valeriy Pantyukhin/
Alexander Bondar

Technical Writers / 4GL Programmer

Last Updated 05 March 2012

4GL Training Guide

Copyright © 2006-2011 Querix Ltd. All rights reserved.

Part Number: 019-006-103-012

Published by:

Querix (UK) Limited. 50 The Avenue, Southampton, Hampshire,
SO17 1XQ, UK

Publication history:

February 2011: Beta edition

March 2011: First edition

March 2012: Updated for Lycia II

Last Updated:

March 2012

Documentation written by:

Technical writers: Elena Krivtsova, Svitlana Ostnek, Valeriy Pantyukhin

4GL programmer: Alexander Bondar

Notices:

The information contained within this document is subject to change without notice. If you find any problems in the documentation please submit your comments by email to documentation@querix.com.

No part of this document may be reproduced or transmitted in any form or by any means, electronic or mechanical, for any purpose without the express permission of Querix (UK) Ltd.

Other products or company names used within this document are for identification purposes only, and may be trademarks of their respective owners.

Table of Contents

INTRODUCTION	1
WHAT IS 4GL?	1
QUERIX-4GL TOOLS	1
<i>The development environment.</i>	1
<i>Ways to launch an application</i>	2
<i>Databases</i>	2
4GL BASICS	3
A BASIC 4GL PROGRAM	4
RESOURCES USED DURING THE DEVELOPMENT.....	4
<i>A 4GL Project</i>	4
<i>A 4GL Program</i>	5
<i>A Source File</i>	5
MAIN... END MAIN BLOCK.....	5
COMMENTS MAKING	5
<i>Comment Indicators</i>	6
<i>Comments Restrictions</i>	6
BASIC SCREEN OUTPUT	6
EXAMPLE.....	7
THE NOTION OF VARIABLES	8
A VARIABLE IN 4GL.....	8
A VARIABLE DECLARATION	8
THE CHAR, VARCHAR, AND STRING DATA TYPES	9
ASSIGNING VALUES	9
ADVANCED DISPLAYING	10
<i>Coordinates</i>	10
<i>Attributes</i>	11
OPERATORS USED WITH CHARACTER VALUES.....	12
<i>The CLIPPED Operator</i>	12
<i>The COLUMN Operator</i>	13
<i>The SPACE Operator</i>	13
<i>Concatenation Operators: Double Pipe and Comma</i>	14
EXAMPLE.....	14
NUMERIC DATA TYPES AND ARITHMETIC EXPRESSIONS.....	18
NUMERIC DATA TYPES	18
<i>Integer Data Types</i>	18

<i>The DECIMAL(p,s) Data Type</i>	19
<i>Fixed-Point Numbers</i>	19
<i>Floating-Point Numbers</i>	19
<i>The MONEY Data Type</i>	20
<i>The FLOAT/SMALLFLOAT Data Types</i>	20
ARITHMETIC OPERATIONS.....	20
THE USING OPERATOR	22
EXAMPLE.....	23
THE SCOPE OF REFERENCE OF VARIABLES.....	26
THE NOTION OF A PROGRAM MODULE	26
<i>A Multi-Module Program</i>	26
TYPES OF VARIABLES WITH REGARD TO THEIR SCOPE	26
<i>Local Variables</i>	27
<i>Module Variables</i>	27
<i>Global Variables</i>	28
<i>Correlation Between Variables of Different Types</i>	29
THE FUNCTION PROGRAM BLOCK.	30
A FUNCTION LIBRARY	32
EXAMPLE.....	33
<i>The GLOBALS File</i>	33
<i>The MAIN Program Module</i>	34
<i>The Function Library</i>	36
FUNCTIONS.....	38
THE FUNCTION DEFINITION AND DECLARATION	38
<i>Formal Function Arguments</i>	38
THE CALL STATEMENT.....	39
<i>Actual Function Arguments</i>	39
THE RETURN STATEMENT AND THE RETURNING CLAUSE.....	43
INVOKING A FUNCTION WITHOUT THE CALL STATEMENT	45
COMPATIBLE DATA TYPES	45
BUILT-IN FUNCTIONS USED WITH CHARACTER VALUES	46
<i>Functions Managing ASCII Code</i>	46
<i>Changing Letter Case</i>	47
<i>Getting the Length of a String</i>	47
<i>Truncating a String</i>	48
THE FUNCTION RETURNING A RANDOM NUMBER	48
VERIFYING THE ARGUMENTS PASSED TO THE APPLICATION	49
<i>Retrieving Individual Arguments</i>	49

<i>Retrieving the Number of Passed Arguments</i>	50
VERIFYING APPLICATION LAUNCH MODE	50
EXAMPLE.....	51
DATE AND TIME VARIABLES	56
DATE VARIABLES.....	56
DATETIME VARIABLES.....	56
<i>The DATETIME Qualifier.....</i>	56
<i>DATETIME Literal</i>	58
<i>DATETIME Character Strings</i>	58
INTERVAL VARIABLES	59
<i>The INTERVAL Qualifier Precision.....</i>	59
<i>The INTERVAL Literal.....</i>	60
<i>INTERVAL Character String.....</i>	61
FORMAT OF DATE AND NUMERIC VALUES.....	61
<i>Date Formats.....</i>	61
<i>Numeric Formats</i>	62
<i>Setting Environment Variables</i>	63
TIME OPERATORS AND FUNCTIONS.....	63
<i>The TODAY Operator.....</i>	64
<i>The CURRENT Operator</i>	64
<i>The UNITS Operator.....</i>	64
<i>The EXTEND() Function</i>	65
<i>The TIME Operator</i>	66
<i>The DATE() Operator.....</i>	66
<i>The DAY(), MONTH(), YEAR(), WEEKDAY() Functions</i>	67
<i>The USING Operator</i>	68
ARITHMETIC TIME EXPRESSIONS	68
DATE AND TIME VARIABLES CONVERSION	69
EXAMPLE.....	70
4GL WINDOWS	75
THE NOTIONS OF THE 4GL SCREEN AND 4GL WINDOWS.....	75
<i>4GL Windows</i>	75
OPENING A WINDOW.....	75
<i>Opening a window with a statement</i>	75
<i>Opening a window with a function</i>	76
<i>Window attributes</i>	77
DISPLAYING TO A WINDOW.....	78
<i>Attributes precedence</i>	78

CURRENT WINDOW	79
CLOSING AND CLEARING WINDOWS.....	81
A 4GL WINDOW AS AN OBJECT	82
<i>Manipulating window objects.</i>	82
METHODS USED WITH A WINDOW VARIABLE	83
<i>Opening a Window</i>	83
<i>Displaying Data to a Window</i>	84
<i>Closing a Window</i>	85
RETRIEVING WINDOW ATTRIBUTES	85
<i>Getting Window Size</i>	85
<i>Getting Window Position</i>	87
CHANGING WINDOW PARAMETERS	88
<i>Setting Window Size</i>	88
<i>Setting Window Title</i>	89
SUSPENDING PROGRAM EXECUTION.....	90
EXAMPLE.....	90
4GL FORM FILES	94
A FORM SPECIFICATION FILE	94
<i>The DATABASE Section</i>	94
<i>The SCREEN Section</i>	95
<i>Comments in a Form File</i>	95
ADDING STATIC OBJECTS TO A FORM	96
<i>Restrictions</i>	97
FORM FIELDS.....	97
OPENING AND DISPLAYING A FORM	98
<i>Form Display Attributes</i>	99
<i>Opening a Form in a Window</i>	100
<i>Position of the Form</i>	100
GRAPHICAL ELEMENTS IN A SOURCE FILE	101
CLOSING A FORM	103
THE FORM DATA TYPE	103
<i>Manipulating Form Objects.</i>	104
EXAMPLE.....	107
<i>Form Files</i>	111
SCRIPT OPTIONS.....	114
A SCRIPT FILE	114
THE STRUCTURE OF A SCRIPT FILE	115
<i>Properties of an Object</i>	116

<i>Wildcard Symbols in an Address</i>	116
THE STRING OBJECT	117
<i>Components with No Form</i>	117
<i>Hiding a String</i>	117
<i>Replacing a Text String</i>	118
THE DEFAULT OBJECT	118
SETTING THE FONT	119
COLOUR PROPERTIES	120
<i>Colour Specification</i>	121
<i>Applying Colours</i>	121
MOVING OBJECTS	123
<i>Moving Objects by a Certain Offset</i>	124
<i>Moving all Objects off a Particular Row</i>	124
<i>Moving an Object to a Specific Row or Column</i>	124
WINDOW STYLE	125
TEMPLATES.....	126
EXAMPLE.....	127
<i>The Form File</i>	128
<i>The Script File</i>	129
CONDITIONAL STATEMENTS AND LOGICAL EXPRESSIONS	132
BOOLEAN VALUES AND EXPRESSIONS.....	132
<i>Relational Logical Operators</i>	132
<i>Boolean Operators</i>	133
<i>Results of Boolean Expressions</i>	133
THE BOOLEAN DATA TYPE	134
THE IF STATEMENT	135
THE CASE STATEMENT	137
<i>The EXIT CASE statement</i>	139
THE TRUE AND FALSE KEYWORDS	140
THE MATCHES AND LIKE OPERATORS	140
THE NULL TEST.....	142
MESSAGE BOXES.....	142
<i>Displaying a Message Box</i>	143
<i>fgl_message_box() Function</i>	144
<i>fgl_winbutton() Function</i>	146
<i>fgl_winquestion() Function</i>	147
EXAMPLE.....	148
PROGRAM WORKFLOW CONTROL.....	156

THE PROMPT STATEMENT	156
THE SLEEP STATEMENT.....	156
FGL_GETKEY() FUNCTION	157
THE GOTO AND LABEL STATEMENTS	157
THE EXIT PROGRAM STATEMENT	159
THE DEFER STATEMENT.....	159
SETTING OPTIONS FOR PROGRAM TERMINATION	159
<i>OPTIONS ON CLOSE APPLICATION</i>	159
<i>OPTIONS ON HANGUP TERMINATE SIGNAL CALL</i>	160
TITLEBAR OPTIONS.....	160
<i>Deactivating the Close Button</i>	160
<i>Deactivating the Maximize and Minimize Buttons</i>	160
EXAMPLE.....	161
<i>The Script File</i>	163
CONDITIONAL LOOPS	164
FOR LOOP	164
<i>The TO Clause</i>	164
<i>The STEP Clause</i>	165
<i>The Declaration Clause</i>	166
<i>The CONTINUE FOR Statement</i>	167
<i>The EXIT FOR Statement</i>	167
WHILE LOOP.....	168
<i>The Declaration Clause</i>	169
EXAMPLE.....	170
USER INTERACTION	176
THE PROMPT STATEMENT	176
<i>The Receiving Variable</i>	176
<i>The FOR Clause</i>	177
<i>The HELP Clause</i>	177
<i>The ATTRIBUTE Clause</i>	179
<i>The ON KEY Clause</i>	179
<i>The BEFORE PROMPT Clause</i>	181
<i>The EXIT PROMPT Keywords</i>	181
<i>The AFTER PROMPT Clause</i>	182
<i>Opening a Prompt in a Separate Window</i>	182
PREVENTING AN APPLICATION FROM CLOSING DURING INPUT	183
THE MENU STATEMENT	184
<i>The COMMAND Block</i>	186

<i>MENU Execution Control Statements</i>	189
<i>The BEFORE MENU block</i>	190
SETTING ICONS FOR THE MENU OPTIONS	192
THE POSITION OF THE MENU AND PROMPT LINES ON THE SCREEN	193
EXAMPLE.....	194
<i>The Form File</i>	202
<i>The Help File</i>	203
<i>Image Files</i>	203
INTERACTIVE FORM ELEMENTS.....	205
FORM FIELDS.....	205
<i>The SCREEN Section</i>	205
<i>The ATTRIBUTES Section</i>	206
<i>The INSTRUCTIONS Section</i>	207
<i>Multi-Segment Fields</i>	209
DISPLAYING DATA TO FIELDS	209
ADDITIONAL FIELD ATTRIBUTES	210
<i>Colour Attributes</i>	210
<i>Conditioned Colour Attributes</i>	211
OTHER WIDGETS	213
<i>The Dynamic Label</i>	213
<i>The Browser</i>	214
<i>The Image</i>	216
TEXT ALIGNMENT	217
A DIALOG BOX FOR MANIPULATING FILES	218
EXAMPLE.....	221
<i>Form Files</i>	225
<i>The Image File</i>	227
SCREEN AND PROGRAM RECORDS.....	228
A PROGRAM RECORD	228
<i>RECORD Members</i>	228
<i>Referencing Record Members</i>	229
A SCREEN RECORD	231
<i>Screen Record Fields</i>	232
<i>Screen records in the INSTRUCTIONS Section</i>	233
DISPLAYING VALUES OF A PROGRAM RECORD TO A SCREEN RECORD	233
THE DISPLAY BY NAME STATEMENT.....	235
THE INITIALIZE STATEMENT	235
A PROGRAMMER DEFINED DATA TYPE.....	236

EXAMPLE.....	237
<i>The Form File</i>	241
<i>The Image File</i>	243
SCREEN AND PROGRAM ARRAYS.....	244
THE PROGRAM ARRAY DECLARATION	244
DYNAMIC ARRAYS	245
REFERENCING ARRAY ELEMENTS.....	248
<i>Referencing Elements of a Dynamic Array</i>	250
THE SCREEN ARRAY	251
DISPLAYING A PROGRAM ARRAY TO A SCREEN ARRAY.....	252
<i>The Binding Clause</i>	252
<i>The SET_COUNT() Function</i>	253
<i>Displaying Array Values</i>	253
<i>Screen Array Colour Script Options</i>	254
THE SCREEN GRID	255
OPTIONAL CLAUSES.....	257
<i>The ATTRIBUTE Clause</i>	257
<i>Scrolling During the DISPLAY ARRAY Statement</i>	258
<i>The WITHOUT SCROLL Clause</i>	258
<i>The BEFORE DISPLAY Clause</i>	259
<i>The EXIT DISPLAY Statement</i>	259
<i>The BEFORE ROW Clause</i>	260
<i>The ON KEY Clause</i>	260
<i>The AFTER ROW Clause</i>	261
<i>The AFTER DISPLAY Clause</i>	261
<i>The CONTINUE DISPLAY Keywords</i>	262
<i>The END DISPLAY Keywords</i>	262
THE BUTTON WIDGET.....	262
<i>The ON ACTION Clause</i>	264
FINDING THE SIZE OF A SCREEN AND A PROGRAM ARRAY AT RUNTIME	266
MANAGING DYNAMIC ARRAYS.....	267
RETRIEVING INFORMATION ABOUT THE CURRENT WINDOW	267
EXAMPLE.....	268
<i>Form Files</i>	280
PERFORMING INPUT.....	283
THE INPUT STATEMENT	283
<i>The Binding Clause</i>	283
<i>The ATTRIBUTE Clause</i>	284

<i>The HELP Clause</i>	285
<i>The INPUT Control Block</i>	285
<i>The Field Order</i>	289
<i>The CONTINUE INPUT Keywords</i>	289
<i>The EXIT INPUT Keywords</i>	290
FACILITATING INPUT	290
<i>Fgl_lastkey()</i>	290
<i>Fgl_keyval()</i>	290
<i>The ERROR Statement</i>	290
INPUT FIELD ATTRIBUTES	290
<i>AUTONEXT</i>	291
<i>CENTURY</i>	291
<i>COMMENTS</i>	291
<i>DEFAULT</i>	293
<i>FORMAT</i>	294
<i>INVISIBLE</i>	295
<i>NOENTRY</i>	296
<i>PICTURE</i>	296
<i>REQUIRED</i>	297
FORM TABS	297
<i>Adding Tabs</i>	299
<i>Switching Between Form Tabs</i>	301
CALENDAR WIDGET.....	302
HOTLINK WIDGET	304
EXAMPLE.....	305
<i>Form Files</i>	314
WAYS OF PRODUCING OUTPUT	319
ERRORS AND MESSAGES	319
<i>The MESSAGE Statement</i>	319
<i>The ERROR Statement</i>	320
FUNCTIONS DISPLAYING HELP	321
<i>The showhelp() Function</i>	321
<i>The fgl_gethelp() Function</i>	324
LARGE DATA TYPES.....	324
<i>The TEXT Data Type</i>	324
<i>The BYTE Data Type</i>	325
INITIALIZING LARGE VARIABLES	326
<i>IN MEMORY</i>	326

<i>IN FILE</i>	326
<i>Freeing Memory Allocated</i>	327
<i>The fgl_copyblob() Function</i>	328
EDITING LARGE VALUES	328
<i>Displaying and Editing Images</i>	329
<i>Displaying and Editing Text</i>	329
EXAMPLE.....	329
<i>The Form File</i>	333
<i>The Help File</i>	334
ADVANCED INPUT OPTIONS	335
THE INPUT BY NAME STATEMENT.....	335
THE INPUT OPTIONS	336
<i>INPUT WRAP/INPUT NO WRAP</i>	336
<i>FIELD ORDER</i>	336
BUILT-IN FUNCTIONS USED DURING INPUT.....	338
<i>The Functions Checking the Fields Data</i>	338
<i>Retrieving Information From the Field Buffer</i>	341
<i>Functions that Return Key Values</i>	345
FUNCTION BUTTON FIELD.....	348
EXAMPLE.....	350
<i>Form Files</i>	360
<i>The Image File</i>	363
FORM WIDGETS FOR RESTRICTED INPUT	364
THE RADIO BUTTON WIDGET	364
<i>Defining the Options and Their Values</i>	365
<i>Triggering Actions</i>	366
THE CHECKBOX WIDGET	367
<i>Defining the Checked and Unchecked values</i>	368
<i>Triggering Actions</i>	369
FUNCTIONS FOR WORKING WITH FORM FIELDS	369
<i>Finding the Position of the Cursor</i>	369
<i>Finding the Name of the Active Field</i>	370
<i>Moving the Cursor to a Definite Position</i>	371
<i>Getting Information About the Current Field</i>	371
SCRIPT OPTIONS	372
<i>Manipulating Field Borders</i>	372
<i>Field Overwrite Script</i>	373
EXAMPLE.....	374

<i>The Form File</i>	377
<i>The Script File</i>	378
INPUT FROM A LIST	381
CREATING A COMBO BOX WIDGET.....	381
<i>The List of Values</i>	381
FUNCTIONS FOR MODIFYING THE DROP DOWN LIST AT RUNTIME	383
<i>Clearing Items from the Combo Box List</i>	383
<i>Counting the Number of Items in the Combo Box List</i>	385
<i>Finding the Position of a Specific List Item</i>	385
<i>Finding the Name of a Specific List Item</i>	386
<i>Adding Items to the List</i>	387
<i>Replacing a Specific List Item</i>	388
<i>Sorting the Combo Box List Entries</i>	390
<i>Restoring the Original List</i>	391
EXAMPLE.....	392
<i>The Form File</i>	396
PERFORMING INPUT FROM A SCREEN ARRAY.....	399
THE INPUT ARRAY STATEMENT.....	399
<i>The Binding Clause</i>	399
<i>The WITHOUT DEFAULTS Keywords</i>	400
THE ATTRIBUTE CLAUSE.....	401
<i>CURRENT ROW DISPLAY</i>	401
<i>COUNT</i>	401
<i>MAXCOUNT</i>	402
THE HELP CLAUSE.....	402
THE INPUT ARRAY CONTROL BLOCK	403
<i>The BEFORE ROW Clause</i>	403
<i>The AFTER ROW Clause</i>	404
<i>The CONTINUE INPUT Keywords</i>	404
<i>The EXIT INPUT Keywords</i>	404
INSERTING AND DELETING ROWS.....	404
<i>Enabling and Disabling the Insert and Delete Keys</i>	405
<i>Control Clauses Influencing Inserting and Deleting of Rows</i>	406
SCREEN ARRAYS WITH GRAPHICAL WIDGETS	408
USING THE BUILT-IN FUNCTIONS FOR MANIPULATING THE INPUT.....	410
<i>The arr_curr() Function</i>	410
<i>The arr_count() Function</i>	411
<i>The fgl_scr_size() Function</i>	411

<i>The scr_line() Function</i>	412
<i>The fgl_dialog_setcurrline() Function</i>	412
EXAMPLE	414
<i>Form Files</i>	425
USING A TOOLBAR	428
CREATING DYNAMIC TOOLBARS	428
<i>The Button Label</i>	429
<i>The Button Icon</i>	429
<i>The ORDER Keyword</i>	430
<i>Field Specific Toolbar Buttons</i>	432
MODIFYING THE DYNAMIC TOOLBAR AT RUNTIME	433
<i>Changing Button Labels</i>	434
<i>Hiding Toolbar Labels and Tooltips</i>	436
<i>Adding Toolbar Dividers</i>	437
EXAMPLE	439
<i>The Form Files</i>	442
<i>Image Files</i>	444
RUNNING COMMANDS	446
PARENT AND CHILD PROCESSES	446
THE RUN STATEMENT	446
<i>Child 4GL Applications</i>	446
<i>The WITHOUT WAITING Keywords</i>	447
RUNNING NON-4GL APPLICATIONS IN GUI MODE	448
<i>Using Built-In Functions to Run Windows Processes</i>	448
<i>Using Built-In Functions to Open Files</i>	449
TERMINATION CODE	450
<i>Values Returned by the EXIT PROGRAM Statement</i>	450
<i>The RETURNING Clause</i>	450
APPLICATION LAUNCHER MENU	451
<i>Specifying the Menu Server</i>	451
<i>Specifying an Application Launcher Menu</i>	452
<i>Populating the Launcher Menu with Options</i>	452
<i>Adding a Submenu</i>	453
<i>Publishing and Executing the Menu</i>	454
<i>Executing Programs</i>	455
<i>The Logic of the Application Menu Performance</i>	456
<i>Using Scripts to Change the Display Mode</i>	457
EXAMPLE	457

<i>The Main Program</i>	458
<i>The Program Launched in GUI Mode</i>	461
<i>The Parent Program Launched in Character Mode</i>	463
<i>The Child Program Launched From within a Parent Program</i>	464
INTERACTION WITH DATABASES	466
DATABASES IN 4GL	467
THE DATABASE STATEMENT	467
<i>The Default Database</i>	468
<i>The Current Database</i>	468
<i>The CLOSE DATABASE Statement</i>	468
CREATING AND DROPPING A DATABASE	468
DATABASE TABLES.....	469
DEFINING TABLE COLUMNS	469
<i>Data Types Used When Declaring Columns</i>	470
<i>The DEFAULT Clause</i>	470
<i>The NOT NULL Constraint</i>	471
COLUMN-LEVEL CONSTRAINTS	471
<i>Defining a Column as Unique</i>	471
<i>Defining a Column as a Primary Key</i>	471
<i>Defining a Column as a Foreign Key</i>	472
<i>The CHECK Condition</i>	473
TABLE-LEVEL CONSTRAINTS	473
NAMING CONSTRAINTS	474
TEMPORARY TABLES.....	475
PRIVILEGES.....	475
<i>Database-Level Privileges</i>	475
<i>Table-Level Privileges</i>	476
GETTING FEEDBACK FROM THE DATABASE.....	477
<i>Checking if the Table Already Exists</i>	477
<i>Retrieving Information about a Column</i>	477
EXAMPLE	480
<i>The Form File</i>	487
POPULATING DATABASE TABLES.....	488
THE LOAD STATEMENT	488
<i>The Loaded File</i>	488
<i>The DELIMITER Clause</i>	490
<i>The INSERT Clause</i>	490
INTRODUCTION INTO THE SELECT STATEMENT	491

<i>The Select List</i>	492
<i>The INTO Clause</i>	492
<i>The FROM Clause</i>	493
<i>The WHERE Clause</i>	494
<i>Additional SELECT Facilities</i>	496
<i>A Table Alias</i>	498
THE UNLOAD STATEMENT	498
<i>The Output File</i>	499
<i>The SELECT Clause</i>	499
<i>The DELIMITER Clause</i>	500
<i>Host Variables</i>	500
SQL INTERRUPT OPTION.....	500
EXAMPLE.....	501
<i>UNL Files</i>	506
MODIFYING DATABASE TABLES	515
MODIFYING THE TABLE STRUCTURE.....	515
<i>Adding a Column</i>	515
<i>Deleting a Column</i>	516
<i>Modifying an Existing Column</i>	516
<i>Modifying Constraints</i>	517
INSERTING ROWS INTO A TABLE	517
<i>The VALUES Clause</i>	517
<i>Inserting Values Selected from Another Column</i>	518
DELETING ROWS FROM A TABLE	519
<i>The WHERE Clause</i>	520
MODIFYING THE EXISTING ROWS.....	520
<i>The SET Clause</i>	520
<i>The WHERE Clause</i>	522
EXAMPLE.....	522
VARIABLES AND FORMS ADJUSTED FOR BEING USED WITH DATABASES	530
VIEWS.....	530
<i>The SELECT Clause</i>	530
<i>The Column List</i>	531
<i>Dropping the View</i>	531
SYNONYMS.....	532
<i>Dropping Synonyms</i>	532
DECLARING VARIABLES LINKED TO TABLE COLUMNS	532
<i>RECORD Variables Linked to Database Tables</i>	533

VALIDATING A VALUE AGAINST A DATABASE COLUMN.....	533
<i>Usage</i>	534
INITIALIZING VARIABLES WITH DEFAULT COLUMN VALUES	534
<i>The Default Values</i>	535
FORM ELEMENTS LINKED TO A DATABASE.....	535
<i>Declaring a Non-Formonly Screen form</i>	535
<i>The TABLES Section</i>	536
<i>Linking Form Fields to Table Columns</i>	536
ADDITIONAL FORM FIELD ATTRIBUTES	537
<i>The VALIDATE LIKE Attribute</i>	538
<i>The DISPLAY LIKE Attribute</i>	538
<i>The VERIFY Attribute</i>	538
THE DEFAULT VALUES AND ATTRIBUTES OF TABLE COLUMNS	539
<i>Setting the Default Volumn Values</i>	539
<i>Setting Column Attributes</i>	541
EXAMPLE.....	542
<i>Form Files</i>	549
THE NOTION OF CURSOR	552
DECLARING A READ-ONLY CURSOR	552
THE CURSOR DATA TYPE	553
<i>The SELECT Statement Associated with a Cursor</i>	553
<i>The SELECT Statement Associated with a Variable of the CURSOR Data Type</i>	554
<i>The OUTER Keyword</i>	554
OPENING A CURSOR	555
<i>Opening a Cursor Associated with a Variable of the CURSOR Data Type</i>	556
FETCHING THE ROWS	556
<i>The INTO Clause</i>	557
<i>Fetching the Rows with a CURSOR Variable</i>	557
<i>The Sequence of Fetching Rows</i>	558
RETURNING INFORMATION ABOUT THE CURSOR.....	559
<i>Getting the Cursor Name for a Variable of the CURSOR Data Type</i>	559
CLOSING AND FREEING THE CURSOR	560
<i>The CLOSE Statement</i>	560
<i>Closing a Cursor Declared by Means of a CURSOR Variable</i>	560
<i>The FREE Statement</i>	560
<i>Releasing Resources Allocated to a Cursor Declared Through a CURSOR Variable</i>	560
EXAMPLE.....	561
<i>The Form File</i>	566

DATABASE TRANSACTIONS	569
THE NOTION OF TRANSACTION	569
TRANSACTION LOGGING	569
<i>Cancelling Transaction Logging for Temporary Tables</i>	569
SPECIFYING THE TRANSACTIONS	570
<i>Starting a Transaction</i>	570
<i>Completing a Transaction</i>	571
<i>Rolling Back Changes</i>	571
LOCKS	572
<i>Locking the Whole Database</i>	572
<i>Locking Tables</i>	572
<i>Releasing the Lock</i>	573
<i>Locking Rows</i>	573
A CURSOR WITH HOLD.....	574
<i>Declaring a Cursor WITH HOLD Using a CURSOR Variable</i>	574
DECLARING AN UPDATE CURSOR	574
<i>The OF Clause</i>	575
UPDATING MULTIPLE ROWS	575
<i>NOTFOUND Condition</i>	576
DELETING MULTIPLE ROWS	577
EXAMPLE.....	578
<i>The Form File</i>	590
<i>The Script File</i>	590
ADVANCED CURSOR USAGE	592
THE FOREACH STATEMENT	592
<i>The INTO Clause</i>	592
<i>The Statement List</i>	592
THE SCROLL CURSOR.....	594
<i>Declaring a Cursor with SCROLL Using a CURSOR Variable</i>	594
PROCESSING A SCROLL CURSOR.....	595
<i>Processing a SCROLL Cursor Associated with a CURSOR Variable</i>	595
ADVANCED SELECT OPTIONS	597
<i>Aggregate Expressions</i>	597
<i>The GROUP BY Clause</i>	598
<i>The HAVING Clause</i>	599
<i>The ORDER BY Clause</i>	599
THE INSERT CURSOR.....	600
<i>The INSERT Statement Associated with the Cursor</i>	600

<i>The Insert Cursor Created by Means of a CURSOR Variable</i>	601
<i>Processing the Insert Cursor</i>	602
<i>Processing an Insert Cursor Associated With a Variable of the CURSOR Data Type</i>	602
ADDING ROWS TO THE INSERT BUFFER	603
<i>Adding Rows to the Insert Buffer with a CURSOR Variable</i>	603
INSERTING THE ROWS INTO THE DATABASE	604
<i>Closing an Insert Cursor</i>	604
<i>Closing an Insert Cursor Associated with a CURSOR Variable</i>	604
<i>Flushing an Insert Cursor</i>	604
<i>Flushing an Insert Cursor with a CUSOR Variable</i>	605
SQL INTERRUPT OPTION	605
EXAMPLE	606
<i>The Form File</i>	618
PREPARED STATEMENTS	620
THE PREPARE STATEMENT	620
<i>The Name of a Prepared Statement</i>	621
<i>The PREPARED Data Type</i>	622
<i>Text of the Prepared Statement</i>	623
PREPARING STATEMENTS WITH KNOWN AND UNKNOWN PARAMETERS	625
<i>Preparing Statements with Known Parameters</i>	625
<i>Preparing Statements with known Parameters Using a PREPARED Variable</i>	626
<i>Preparing Statements with Parameters Received at Runtime</i>	627
<i>Preparing Statements with Unknown SQL Identifiers</i>	627
REFERENCING PREPARED STATEMENTS WITH PLACEHOLDERS	628
<i>Referencing a SELECT Statement with Placeholders</i>	628
<i>Referencing a SELECT Statement with Placeholders when Using a PREPARED Variable</i>	629
<i>Referencing an INSERT Statement with Placeholders</i>	630
<i>Referencing an INSERT Statement with Placeholders by Means of a PREPARED Variable</i>	631
THE EXECUTE STATEMENT	631
<i>The INTO Clause</i>	632
<i>The USING Clause</i>	633
EXECUTING A PREPARED STATEMENT ASSOCIATED WITH A VARIABLE OF THE PREPARED DATA TYPE	633
THE FREE STATEMENT	634
<i>Freeing Resources Allocated to a Statement Prepared with a PREPARED Variable</i>	634
PREPARING SEQUENCES OF MULTIPLE SQL STATEMENTS	634
STATEMENTS THAT CAN, MUST, AND MUST NOT BE PREPARED	635
<i>Statements that Must be Prepared</i>	635
<i>Statements that Cannot be Prepared</i>	636

SQL... END SQL.....	636
EXAMPLE.....	637
CONSTRUCTING A QUERY BY EXAMPLE	647
<i>The CONSTRUCT Variable Clause.....</i>	648
<i>The ATTRIBUTE Clause.....</i>	650
<i>The HELP Clause.....</i>	650
<i>The CONSTRUCT Input Control Blocks.....</i>	650
<i>The CONTINUE CONSTRUCT Statement</i>	655
<i>The EXIT CONSTRUCT Statement</i>	656
<i>The NEXT FIELD Clause.....</i>	656
<i>User-Entered Search Criteria.....</i>	656
<i>Editing During the Search Criteria Input</i>	657
APPLYING QUERIES BY EXAMPLE	657
EXAMPLE.....	658
<i>Form Files.....</i>	663
MISCELLANEOUS.....	666
WEB SERVICES.....	667
THE NOTION OF WEB SERVICES	667
<i>What is a Web Service?</i>	667
<i>Core Elements Behind a Web Service.....</i>	668
WHAT YOU NEED TO USE WEB SERVICES.....	668
CREATING A WEB SERVICE OPERATION	669
<i>The RETUNRS Keyword</i>	669
COMPILING AND DEPLOYING A WEB SERVICE.....	672
<i>Using Tomcat 7.*</i>	676
ACCESSING AN EXTERNAL WEB SERVICE	676
<i>Finding a URI of an External Web Service.....</i>	676
<i>Generating a Meta Data Description File and Web Service Client Stub</i>	676
<i>Explaining the Stub Syntax</i>	679
MAKING IT ALL WORK	681
EXAMPLE.....	682
<i>Web Service</i>	683
<i>Client</i>	686
THE NOTION OF REPORTS.....	694
THE GENERAL NOTION OF REPORT	694
THE REPORT PROGRAM BLOCK	694
<i>A Simplest Case of the REPORT Program Block.....</i>	695
REPORT DRIVER	696

THE START REPORT STATEMENT.....	697
<i>The TO Clause</i>	698
<i>The WITH Clause</i>	700
OUTPUT TO REPORT KEYWORDS AND CONDITIONAL LOOPS	702
FINISH REPORT AND TERMINATE REPORT STATEMENTS	703
EXAMPLE.....	704
FORMATTING REPORT OUTPUT	708
THE DEFINE SECTION.....	708
THE FORMAT SECTION	709
<i>FORMAT SECTION Control Blocks</i>	710
<i>Statements in the Format Control Blocks</i>	710
THE OUTPUT SECTION	716
THE ORDER BY SECTION	717
<i>The Sorting List</i>	718
<i>The EXTERNAL Keyword</i>	719
EXAMPLE.....	719
<i>The Function Library</i>	725
STATEMENTS USED IN REPORT SECTION.....	733
THE PRINT STATEMENT	733
<i>The Output Character Position</i>	734
<i>The FILE, TEXT, and BYTE Keywords</i>	734
<i>Expressions and Operators Used with the PRINT Statement</i>	735
OPERATORS USED IN THE PRINT STATEMENT	735
<i>The ASCII Operator</i>	735
<i>The COLUMN Operator</i>	735
<i>The SPACE Operator</i>	736
<i>The LINENO Operator</i>	737
<i>The PAGENO Operator</i>	737
<i>The WORDWRAP Operator</i>	738
AGGREGATE REPORT FUNCTIONS	739
<i>Aggregate Dependencies</i>	739
<i>Calculating Average and Total Values</i>	740
<i>Calculating the Number of Records</i>	741
<i>Calculating Maximum and Minimum Values</i>	741
EXIT REPORT STATEMENT	742
THE NEED STATEMENT.....	742
THE PAUSE STATEMENT	742
<i>The PAUSE Statement on Linux/UNIX</i>	743

<i>The PAUSE Statement on Windows</i>	743
THE SKIP STATEMENT.....	743
EXAMPLE.....	744
<i>The Function Library</i>	752
ERROR HANDLING.....	753
THE NOTION OF ERRORS AND WARNINGS	753
<i>The Status Built-In Variable</i>	754
THE WHENEVER STATEMENT	755
<i>The ERROR Event</i>	755
<i>The NOT FOUND Event</i>	757
<i>The WARNING Event</i>	757
<i>The ARGUMENT ERROR Event</i>	758
<i>The RETURN ERROR Event</i>	758
<i>The UNDEFINED FUNCTION Event</i>	758
<i>The QUIT Event</i>	758
<i>The INTERRUPT Event</i>	758
<i>The TERMINATE Event</i>	759
<i>The HANGUP Event</i>	759
<i>The FATAL ERROR Event</i>	759
THE ACTIONS APPLIED TO THE TRAPPED EVENTS	759
<i>The GOTO Statement</i>	759
<i>The CONTINUE Statement</i>	759
<i>The STOP Statement</i>	759
<i>The RAISE Statement</i>	759
<i>The CALL Statement</i>	760
<i>The Scope of the WHENEVER Statement</i>	761
THE TRY... CATCH STATEMENT	761
ERROR HANDLING USING THE SQLCA RECORD.....	764
RETRIEVING THE ERROR MESSAGE	766
WORKING WITH AN ERROR LOG FILE	768
<i>Opening or Creating an Error Log File</i>	768
<i>Adding Comments to the Error Log File</i>	770
ACCOMPANYING AN EVENT WITH A SOUND SIGNAL.....	771
RETRIEVING ERRORS FROM A NON-INFORMIX RDBMS	772
<i>Getting an Error Code from a Non-Informix RDBMS</i>	772
<i>Retrieving the Text of a Native RDBMS Error Message</i>	773
FINDING OUT THE LAST ERROR RETURNED BY THE DATABASE INTERFACE	774
EXAMPLE.....	775

<i>The Form File</i>	784
INDEX	785



Introduction

This document is intended for 4GL learners not acquainted with 4GL or SQL. It is a step-by-step guide based on 4GL code examples which can be used to learn the 4GL by a person without any background programming knowledge.

What is 4GL?

The 4GL is fourth-generation programming language easy to study and use. This language uses keywords which very much resemble the English language, thus it is easy to understand and memorize. The 4GL has been designed to reduce programming effort, the time it takes to develop software, and the cost of software development.

Informix-4GL is a programming language developed by Informix during the mid-1980s. It includes such elements as embedded SQL, a report writer language, a form language, statements and functions. Some other companies created their own tool on the basis of Informix-4GL with some extended functionality and language modifications, such as graphical user interfaces and integrated development environments. Querix also created a number of tools on the basis of Informix-4GL with some advanced language functionality contributing to its flexibility and usability.

The 4GL is mainly used for working with relational databases. The Informix-4GL was created to work with the Informix database, but Querix has adapted it to work with different types of databases (Oracle, MySQL, etc.). The main function of 4GL is to create a user interface for working with a database: viewing and editing its contents. With the help of 4GL it is also possible to sort the information received from the database, to modify tables, to receive the input from a user and so on.

Querix-4GL Tools

Querix created a set of tools for working with 4GL. These are the compiler with a Studio which allows a programmer to develop 4GL applications with comfort and speed, the graphical thin clients which run 4GL applications in the graphical mode which imparts the modern look and feel to the applications, and others.

The development environment

Lycia Studio is the development environment which comprises a number of tools used for:

- Creation, edition, and compilation of 4GL source files and form files.
- Creation and manipulation of auxiliary files (image files, script files, etc.)
- Detailed debugging of applications
- Working with the resources from repositories (i.e. CVS repository)
- Working with different types of databases

For more information about the Studio, see the "Lycia Getting Started" and "Lycia Developers Guide" documents. They can be downloaded from the Querix web site or found in your Lycia documentation package. You can also use the interactive help system from within the Lycia Studio; it is invoked by pressing F1 or it is available from the Help menu option.



Ways to launch an application

There are thin clients used for launching 4GL applications (i.e. LyciaDesktop, Phoenix, Chimera and others). They are very powerful and extensible tools for rendering your 4GL applications in familiar working environments. Phoenix and LyciaDesktop will render 4GL applications in a Microsoft Windows™ style environment. Chimera is a java-based client which works on both Microsoft Windows™ and UNIX/Linux platforms. The applications launched using a thin client can use media files, such as images, Phoenix and Chimera can also use the script files which customize the feel and look of the application, whereas LyciaDesktop is using .xml style sheets for this purpose.

Both graphical clients use the application server to which the applications are deployed before launching. An application server can be located on the same machine where the application has been compiled, or on a different one. Thus you can launch applications on a remote machine.

It is also possible to launch an application in a character mode via Studio, this does not require any thin clients or application servers.

For more information about the thin clients and the application servers see "Graphical Clients Reference", "Phoenix Developers Guide", "Chimera Developers Guide", "LyciaDesktop Developers Guide", and "Lycia Developers Guide" documents. They can be downloaded from the Querix web site.

Databases

Querix tools allow your 4GL applications to work with a number of different database types. Besides the Informix database, one and the same application can be run against Oracle database, SQL Server database and others. You can instruct your applications to connect to a different type of a database just by selecting the database type from the list. Thus if you have two databases with the same name and contents but of different types (e.g. an Informix database and an Oracle database), your application will connect to one of them depending on the database type currently selected in the Studio.

For more information about the databases and connecting to them see the "Lycia Developers Guide", "Lycia Getting Started Guide", individual database migration guides and the user documentation of the corresponding database vendor.



4GL BASICS

This section will teach you to create simple and more complicated 4GL programs which include most of the basic 4GL functionality. They will include different means of displaying information and a number of ways to receive the input from the user. To be able to compile and run programs contained in this section you need nothing except your Lycia compiler and at some point a graphical thin client.



A Basic 4GL Program

Lycia Studio works with a special location on your machine called the workspace. You specify the folder which you want to be your workspace when Lycia Studio starts. The workspace is used to contain all the resources associated with the process of application development. You can have several workspaces and you can switch between them as you work.

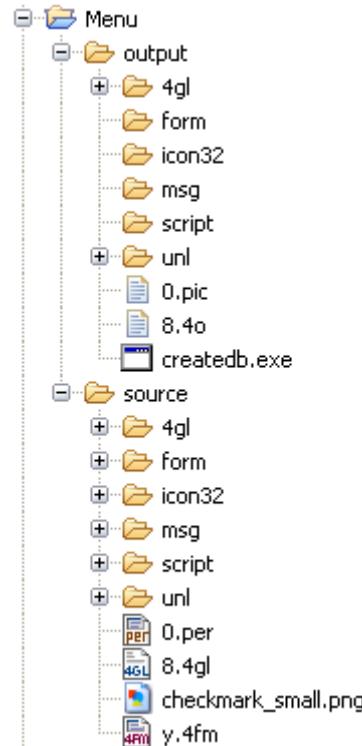
Resources Used During the Development

In Lycia Studio the process of development is guided by the concept of projects. These are the largest units into which the development environment is divided. In their turn the projects include programs and source files.

A 4GL Project

A project is a set of resources such as source files, programs and any additional data which serve one purpose. Each project (either imported or created in Lycia Studio) must be located in a separate folder within the workspace. Technically a project is a folder which contains the build targets and the compilation results received from building these targets. Typically the source files are located in the "source" subfolder and the compilation results are located in the "output" subfolder. A project can include one or more programs which can interact with each other or work independently.

The image below illustrates how a project "Menu" is organized in the Studio:





A 4GL Program

A 4GL program is a set of commands, written in the 4GL language. The program may consist of one or more files, containing source code, screen forms, messages, media data, or scripts. Technically a program is a compiled .exe executable file (or other executable, depending on the platform) which appears in the output folder after building. In the Studio there is an object called a program, it is displayed in the 4GL Project view regardless of whether the program has actually been built and is used to indicate all the resources associated with this program. For a program to build successfully all the resources associated with it must be built also.

A Source File

Each program must have at least one file containing the source code associated with it. A file, containing source code has .4gl extension. It contains statements, functions and other code (or keywords), which direct the program to perform specified actions and refer the program to other program files. All the keywords in 4GL are not case sensitive and can be written in the upper or lower case, but for the sake of code readability they will be capitalized in the examples and in the text.

The statements are organized into larger units, called *program blocks*. There are three kinds of program blocks: MAIN, FUNCTION and REPORT. Each block should begin with a keyword, which corresponds to the name of the block, and end with the END keyword, followed by the block name (END MAIN, END FUNCTION, END REPORT).

When you create a 4GL program, you can put statements and comments into one or more source code files. These source code files are called modules.

A program needs to be compiled. For the information about how to compile and run a program see "Lycia Developers Guide" and "Lycia Getting Started".

MAIN... END MAIN Block

The MAIN program block is a block which typically calls functions and reports to do the work of the application.

The statements, located between the MAIN and END MAIN keywords comprise the MAIN block. The MAIN block does not need to be called; it is executed automatically, when the program is started. The program execution begins from the first statement following the MAIN keyword and ends when the END MAIN statement is executed, unless something prevents the program from completing.

The MAIN block should be the first program block of the module where it is located. Each program can contain only one MAIN block, even if the program has several modules. It can't be located within any other program blocks or functions as well as no other blocks can be located inside it.

Comments Making

When writing a program, you may need to make some comments for future reference. These comments may include some notes, descriptions, explanations or code snippets – any additional information you or other people will need to understand the program structure and purpose. The comments do not influence the source code; they are ignored during the compilation and are not displayed at runtime. They can be viewed only while editing the source code. In Lycia Studio the commented lines are typically light green.



Comment Indicators

You can put comments into any part of the source code, but you should indicate them, otherwise the program will designate them for parts of the source code and will produce a compile-time error. There are three types of comments indication:

- The # ("pound", or "sharp") symbol indicates a one-line comment that starts right after this sign and continues only to the end of the line. Any symbols placed on this line to the right of this symbol are treated as comments. No closing symbol is required.
- Double hyphens or minus signs (--) works the same as the # symbol comment indicator.
- The double-brace symbols indicate a multiple-line comment. The comment begins right after the left-brace ({) and ends with the right-brace (}) closing symbol. These symbols can be placed on the same line or on different lines. All the symbols between the double-brace symbols become commented regardless of the number of lines.

The comment indicators may be very useful during program development. You can indicate some functions and statements as comments in order not to delete them from the code. You can easily put them back to the program by deleting the comment indicators.

	Note: In the further chapters of this tutorial we will use the comments within the code samples to point out the purpose and effect of some 4GL structures.
--	--

Comments Restrictions

When creating comments, keep in mind the fact that a number of restrictions exists concerning the use of comment indicators.

- If you use a comment indicator within a quoted string (e.g. in a DISPLAY statement), 4GL will recognize it as a part of that string, but not as a comment indicator.
- It is impossible to nest comments within comments by using left- and right-brace symbols.

Basic Screen Output

One of the main means of communication between a program and a user is the DISPLAY statement. This statement makes the program display character strings, values of variables, forms, arrays, media files on the screen. The DISPLAY keyword may be followed by different clauses which determine how and where on the screen the information will be displayed. They will be discussed in the following chapters.

One of the simplest examples of the DISPLAY statement usage is displaying a character string, which contains a message. A character string is a sequence of characters enclosed in quotation marks. These can be alphabetic characters, numbers, punctuation marks, and other symbols. The string may be rather long, but if its size is bigger than the width of the screen or window, a part of the string will not be displayed.

If you want to display a character string on the screen, you should put it immediately after the DISPLAY keyword. Remember, that the string should begin and end with quotation marks ("") or double quotes (" "), which will help the 4GL to distinguish it from the source code.

```
DISPLAY "Hello, world!"
```

This code line will display *Hello, world!* to the screen console. The output will be different depending on whether the program is run in character mode or in graphical mode.



- In character mode it is displayed to the screen and, if there is more than one DISPLAY statement, the character strings displayed by them appear one under another in a kind of a list.
- In graphical mode it is displayed to the special console window which is opened automatically.

The whole program displaying a character string will have the following structure:

```
MAIN
    DISPLAY "Hello, world!"
END MAIN
```

You can also add a comment to your program. Here we add a comment line before the MAIN block, but it can occur anywhere within the program:

```
#This is my first program
MAIN
    DISPLAY "Hello, world!"
END MAIN
```

If you need to have quotation marks in your message, put a backslash (\) before each quotation mark, or use different kinds of quotation marks to indicate the character string and the quotations to be displayed:

```
DISPLAY "Type 'Y' if you want to exit the program"
DISPLAY 'Type "Y" if you want to exit the program'
DISPLAY "Type \"Y\" if you want to exit the program"
```

Remember, that you can't use the same quotes, because the program will recognize them as the end of the character string.

Example

Here is an example of an elementary 4GL program.

```
#####
#This is the simplest 4GL program
#####

MAIN -- the beginning of the MAIN block

    DISPLAY "Hello, world!" -- the displayed value

END MAIN -- the end of the MAIN block
```



The Notion of Variables

A Variable in 4GL

A 4GL variable is a named value container of certain data type. The *value* of a variable is the data, stored in it. The data stored in the variable can be changed throughout program execution. Variables have identifiers by which they are referenced by the program.

The variable identifier (name) may contain one or several characters, but it must not contain blank spaces, punctuation marks, symbols, except the underscore (_) symbol. A variable name mustn't also consist of digits only or begin with a digit. Each variable must have its unique name, and this name must not match any of the *reserved words* (see IBM Informix 4GL Reference Manual, Appendix G Reserved words).

A Variable Declaration

A variable must be declared before it can be used by your program. To declare a name and a data type of a variable, use the DEFINE statement. The DEFINE statement can be followed by one or more declaration blocks, separated by commas. The declaration block consists of one or several variable names, separated by commas and followed by the data type keyword. Sometimes, it may be possible or necessary to define the size of a variable (it should be given in brackets after the data type declaration):

```
DEFINE a, b CHAR(10)
```

There can be one or more DEFINE statements in one program module, but all of them must be located at the very beginning of the program block before any executable statements.

```
MAIN
DEFINE a, b CHAR(10),
      c VARCHAR
DEFINE e INTEGER
...
END MAIN
```

You can add the OF keyword between the variable identifier and the data type specification. This keyword is optional and has no effect on the result of the data type declaration. However, the keyword can make the source code more convenient to read. The following example has the same effect as the example given above:

```
MAIN
DEFINE a, b OF CHAR(10),
      c OF VARCHAR
DEFINE e OF INTEGER
...
END MAIN
```



The CHAR, VARCHAR, and STRING Data Types

In the previous chapter we discussed how a character string may be displayed on the screen. This isn't the only way to do it. You can also display some character values stored in variables of CHAR, VARCHAR, and STRING data types.

The CHAR (or CHARACTER) data type is a data type used to store a character string. The string can consist of any printable symbols, including punctuation marks, symbols, and digits.

The length of the string is determined by the maximum *size* specified in brackets, when the data type is declared. The value assigned to the variables can be smaller than the size, but cannot be larger than that. The minimum size of CHAR is 1 byte, and this is the default maximum size of a variable of this type. The default variable size is applied when no size is specified when declaring the variable. The maximum size that can be specified for a CHAR variable is 32,767 bytes. Usually one byte is equal to one character (in the default system locale).

A variable defined as CHAR(20) may contain 20 or fewer bytes, but it cannot contain more than 20 bytes. If a variable size is shorter than it is declared, blank spaces are added to it until its size is equal to the declared one.

The VARCHAR data type is similar to the CHAR data type. The difference is that the character string length may vary. You can declare the maximum size of a VARCHAR variable using the *(size)* parameter. The size of the VARCHAR variable is limited only by the storage capacity of your system, thus a VARCHAR value can be larger than a CHAR value. If you don't do this, the default size will be set to 1.

If a VARCHAR variable length is shorter than it is declared, no blank spaces are added to the variable value.

A variable of the STRING data type is a variant storage for character strings. The size of a STRING variable varies depending on the length of the value stored in it and you do not need to specify its maximum size explicitly. The upper limit for the size of a STRING variable 4000 characters. If no value is stored in a STRING variable, it has a length of one character. If a value is assigned to it, it takes as many bytes to store the value as many characters are in it. If another value is assigned to the same variable afterwards, its size changes in accordance with the number of characters in that value.

The difference between a character string which has been assigned to a CHAR/VARCHAR variable and a character string used directly without being assigned to any variable, is that you have to print the latter in each portion of the code where you want to display it or to refer to it. Such character string isn't stored in the program memory and can't be referred to without being typed from the first to the last symbol in the corresponding part of the program code. Whereas a variable refers to a character string stored in memory and such character string needs not to be retyped each time.

	<p>Note: Usually, the CHAR, VARCHAR, and STRING data type need one byte for per character, therefore, the size of the value in bytes is equal to the number of characters stored in the variable. However, in some Eastern locales these data types may require more than one byte per character, even if it is a white space character.</p>
--	---

Assigning Values

When a variable is declared with the help of the DEFINE statement, a program allocates memory space to it, and its value is set to NULL. This means that a variable has no value yet.



A variable should be initialized (have an assigned value) before you refer to it in your program. The LET statement is the most common way of variable initialization. When 4GL executes the LET statement, it evaluates the right part of the statement (the one after the (=) sign) and assigns its value to the variable on the left of the equal sign. There can be only one expression following the LET statement, so if you want to initialize several variables, you need to use a separate LET statement for each variable:

```
DEFINE a,b,c CHAR(10)
LET a = "string 1"
LET b = 'string 2'
LET c = "string 3"
```

Note, that you should give the character values in quotation marks in the same way as you specify a character string; otherwise the program will recognize them as other variables or as parts of your program code. If you want the quotation marks to be a part of the CHAR or VARCHAR value, use a backslash (\) before each of them or use quotation marks of a different type.

You can also assign a value of one variable to another one. To do this, place the variable, the value of which you want to assign, to the right of the equal sign:

```
LET a = b -- assigning value of variable b to variable a
```

Sometimes it is necessary to change a part of a character string, assigned to a variable or assign only a substring of the value of one variable to another one. You can do this by identifying a substring of a CHAR or a VARCHAR variable. The substring is identified in square brackets following the variable identifier. Inside the square brackets you can specify the first and the last characters of the substring you want to change or assign. The following expression:

```
LET a[3,5] = "ABC"
```

means, that "ABC" character string will replace the third, the fourth and the fifth characters of the value of the variable *a*, and the other characters will remain unchanged. You can replace only one character of the string by putting its number into the square brackets (LET a[5]="Z"). You can't replace one character with two or more characters.

The following example assigns a substring of the value of variable var1 to another variable – var2:

```
LET var2 = var1[15,25]
```

Advanced Displaying

We have already discussed the simplest form of the DISPLAY statement which displays a character string to the 4GL screen. In this case, the output is performed to the bottom line of the 4GL screen. Each next line will be displayed below the previous one, moving it upwards.

There are several clauses and key words that allow you to control the way and the place on the screen where the information is displayed.

Coordinates

If you want the information to be displayed at a certain place on the screen or in the current window, you should use the AT clause of the DISPLAY statement. The structure of the DISPLAY ATstatement is as follows:



`DISPLAY display_value AT line, column`

Line and *column* are integers or any variables or expressions which return integer numbers. They specify the place on the 4GL screen or current window, where you want to display your information. If you use the AT clause, you should specify both *line* and *column*.

```
DISPLAY my_char AT 2,5 -- the value of the variable my_char is displayed
-- at the 5th column of the 2nd line
```

```
DISPLAY "character string" AT 3,5 -- the string is displayed
-- at the 5th column of the 3rd line
```

When you specify the coordinates for the DISPLAY statement, keep in mind the size of the variable or string that is to be displayed. The value that can be displayed from the very beginning of the screen, probably, won't fit the screen size, if you display it starting from the middle of the line. If the declared display coordinates exceed the screen limits, an error will occur during the program execution.

Attributes

If you use the AT clause in the DISPLAY statement, you can also use the ATTRIBUTE (ATTRIBUTES is a synonym) clause to change the colour, the colour intensity and the type face of the text displayed. The ATTRIBUTE clause syntax is as follows:

```
ATTRIBUTE (attribute [,attribute])
```

There must be at least one attribute in the parentheses. The attributes are special keywords; they are listed in the table below:

Type	Attribute	Display Result
Colour	WHITE	White colour (default font colour)
	YELLOW	Yellow colour
	RED	Red colour
	MAGENTA	Magenta colour
	BLUE	Dark-blue colour
	GREEN	Green colour
	CYAN	Cyan colour
	BLACK	Black colour
Intensity	NORMAL	Default colour with no intensity applied
	DIM	The colour of the displayed value becomes pale
	BOLD	The colour is intensified
Type face	UNDERLINE	This attribute underlines the displayed values.
	REVERSE	This attribute displays the values in reversed video. The current colour of the font becomes the colour of the background and the colour of the background becomes the colour of the font.
	BLINK	This attribute makes the displayed data blink
	INVISIBLE	This attribute prevents the corresponding 4GL statement from displaying information on the screen



When using the ATTRIBUTE clause, you should remember several restrictions:

- You can use only one attribute specifying the text colour; several colour attributes will be in conflict.
 - The DIM and BOLD attributes have opposite effect and are mutually exclusive.
 - The NORMAL attribute returns the default colour value. Any colour attribute preceding the NORMAL attribute will be ignored; if the DIM or BOLD attributes precede the NORMAL attribute, they will be enabled.
 - You can put the UNDERLINE, REVERSE and BLINK attributes simultaneously to one ATTRIBUTE clause.
 - If the INVISIBLE attribute is used, you won't be able to see the displayed information.
 - If you include several attributes that are in conflict, 4GL will use the one you specified the last.
- . The following is the example of a DISPLAY AT statement with the ATTRIBUTE clause:

```
DISPLAY "Do you want to continue?" at 2,2
ATTRIBUTE (YELLOW, BOLD, UNDERLINE)
```

If you insert these lines to your program code, the program will display "*Do you want to continue?*" in yellow, bold, and underlined.

	Note: You can't use the ATTRIBUTE clause of the DISPLAY statement without using the AT clause. If you do so, the program will produce a compilation error.
--	---

Operators Used with Character Values

There are a number of operators which are typically used with character values. They serve for different purposes such as truncating trailing spaces, converting values or concatenating them. Below you can find the description of the most commonly used operators.

The CLIPPED Operator

When the size of a CHAR variable value is smaller than specified during the variable declaration, the program automatically adds blank spaces to the variable value in order for it to fit the declared size. The variable value is displayed together with all the blank spaces added by the program. It means, that if the variable *my_char* is of CHAR(10) data type and its value is "Yes", the string "Yes" will be displayed together with seven blank spaces following it: <Yes >.

Such display form is sometimes not very convenient, so there is a way to display only the value assigned and no additional blank spaces. The CLIPPED operator returns the value of the variable without additional spaces, but it does not affect the value itself. The CLIPPED operator can be used as follows:

```
DISPLAY my_char CLIPPED AT 2,5
```

The CLIPPED operator should follow each variable you want to be displayed without additional blank spaces. Variables of the VARCHAR data type don't need the CLIPPED operator, because they don't have additional blank spaces. The comma separating two displayed variables should be put after the CLIPPED operator:



```
DISPLAY my_char CLIPPED, my_char2 CLIPPED AT 2,5
```

This part of the program code will display "my_charmy_char2" to the screen. If you don't want the displayed values to look like a single string, you may display a comma or a blank space between them:

```
DISPLAY my_char CLIPPED, " " my_char2 CLIPPED AT 2,5
```

This will result in a following line: "my_char, my_char2".

The COLUMN Operator

The second way to separate the values you want to display is using the COLUMN operator. The COLUMN operator shifts the display value which it precedes by the specified number of characters. The COLUMN operator must be followed by an integer or an expression returning an integer value which reflects the number of characters beginning from the left side of the screen. The value following the COLUMN operator will be displayed on the character following the specified number of characters. It does not take into account the values displayed prior to the operator. Thus, if you want to use this operator, you should manually calculate the number of characters occupied by the previously displayed values and use the number of characters greater, than the preoccupied section, so that the values do not overlap.

```
DISPLAY "first displayed value", COLUMN 30, "second displayed value" AT 5,5
```

In the example above the first displayed value needs at least 21 character to be displayed. You should not forget that it will start on the fifth character and not on the first character of the line (as specified in the AT clause). Therefore, to calculate the COLUMN value so that the second value does not start on the first one you should calculate the following:

(previous value characters)+(left offset)+(required space between value 1 and value 2) = COLUMN *integer*

In our case it will have the following implementation:

$$21+5+1=27$$

So, we need at least 27 characters prior to the second value, if we want only one whitespace to separate the values. To be on the safe side, we used COLUMN 30 – this will increase the distance between the values. Keep in mind that you should leave enough space for any value to be displayed, so do not specify the last characters of a window or the screen in the COLUMN operator, otherwise the values displayed after it will not be visible.

The SPACE Operator

You can also use the SPACE operator to add blank spaces between the values displayed by the DISPLAY statement. The SPACE operator must be preceded with a whole number which specifies the number of blank spaces that are to be inserted to the current character position. The effect of the *number* SPACE operator is the same as the effect of adding a quoted string containing the *number* of blank spaces. The following two lines of the source code will produce the same display result:

```
DISPLAY "first displayed value", 5 SPACE, "second displayed value"  
DISPLAY "first displayed value", "      ", "second displayed value"
```



The second quoted string in the second example consists of five blank spaces.

The main difference between the COLUMN and the SPACE operators is that the COLUMN operator provides static location for the next displayed value, and it is possible, that the long values displayed from variables overlap or, vice versa, are placed at the big distance from each other. The COLUMN operator is useful when you know exactly the maximum length of the displayed values and when you want the display result to look structured: it is easy to display values below each other using this operator.

The SPACE operator allows you to add convenient distances between the values. However, you should keep in mind the possible length of the displayed line and the number of characters that are displayed before the SPACE operator in order to prevent the values of the last variable from being truncated from right.



Note: If the SPACE operator follows a CHARACTER value which is shorter than the declared length of the variable and you do not use the CLIPPED operator, the SPACE operator will add blank spaces to the blank spaces which were used to make the value fit the declared variable size.

The SPACE operator can also be used in a LET statement to assign blank spaces to a value. The effect is the same as assigning a number of quoted blank spaces, but in such a way you need not to count the whitespaces when typing the value to be assigned. In this case the SPACE operator needs to be enclosed in parentheses. The example bellow adds 6 blank spaces to the beginning of the character value:

```
LET a = (6 SPACE), "=zip"
```

Concatenation Operators: Double Pipe and Comma

The double pipe (||) and comma (,) operators are used to concatenate several values into one value. They are typically used with character values. You can assign a concatenated value consisting of a number of variables or quoted character strings to another variable in the following way:

```
LET var1= "this "
LET var2= "is a "
LET var_full1 = var1, var2, "string."
LET var_full2 = var1 || var2 || "string."
```

They function similar in most cases and in the example above *var_full1* and *var_full2* will have the same value which will be "This is a string".

If at least one operand of a concatenated expression has NULL value, using the double pipe (||) symbol for concatenation will return NULL for the whole expression. If you use comma in such case, NULL is returned only if every operand is NULL.

Example

```
#####
# This is an example of declaring character variables and assigning values
# to them.
#####
MAIN
```



```
# The DEFINE statement here declares the variables used by the application;
# variables which are not declared, cannot be used.
DEFINE

    # Three variables are declared to be of CHAR data type
    v1_char      CHAR,           -- A CHAR variable without (size) has the
                                -- default length of 1 byte

    v2_char      CHAR(20),        -- This variable can contain up to 20
                                -- bytes(characters)

    v_max_char   CHAR(32767)       -- This variable can contain up to 32767
                                -- bytes(characters) which is the maximum
                                -- possible size

# The next two are declared to be VARCHAR data type. The size of
# these variables is declared using the principles of the CHAR variables.
# The OF keyword is optional
DEFINE v2_varchar OF VARCHAR(25),

    v_max_varchar OF VARCHAR(32767), -- the VARCHAR data type has the same
                                -- maximum size as a CHAR variable

    v_string     OF STRING         -- the size is not declared and
                                -- depends on the value assigned to it

#####
# Part one: assigning values
#####

# This DISPLAY statement has a defined position and display attributes:

DISPLAY "The uninitialized variable below has NULL value:"
AT 1,2           -- the position of the displayed string
ATTRIBUTE(GREEN, BOLD) -- display attributes assigned to the string

# This DISPLAY statement has only position but does not have attributes:
DISPLAY "v1_char  CHAR = ",v1_char," (the variable has no value)" AT 2,2
SLEEP 3
# Then we use the LET statement to assign some values to the variables

LET v1_char = 'A' -- we assign a string with single quotation marks

LET v_max_char = "1234567890ABCDEFGHIJabcdefghij_-",
      "=():;!:?`|^*#&$@^,.<>"" -- we assign a double
                                -- quoted string

LET v2_varchar = v_max_char -- in such way you assign the values
                            -- of one variable to another one.
                            -- However, only the first 20 symbols are
                            -- assigned to the variable, while its
                            -- length is declared as CHAR(20)
```



```
LET v_string = "This is a string variable with size 38"

DISPLAY "Here are the values assigned to the character variables:"
      AT 4,2 ATTRIBUTE (GREEN, BOLD)

DISPLAY "v1_char      CHAR      = ",v1_char AT 5,2
DISPLAY "v_max_char   CHAR(32767) = ",v_max_char CLIPPED AT 6,2
      -- here the CLIPPED attribute is used
      -- to truncate the blank spaces at the end of
      -- v_max_char variable.

DISPLAY "v2_varchar   CHAR(25)    = ",v2_varchar AT 7,2
DISPLAY "v_string      STRING     = ", v_string AT 8,2
SLEEP 3
DISPLAY "And here is how the quotation marks are used:"
      AT 10,2 ATTRIBUTE (GREEN, BOLD)

LET v_max_varchar = "Inside a string enclosed in double quotes,",
                  "double quotes (") should be duplicated, ",
                  "to be displayed"
      -- the same is true for single
      -- quotation marks

DISPLAY v_max_varchar AT 11,2
SLEEP 3

#####
# Part two: operators
#####

DISPLAY "Usage of the Substring [ ] operator:"
      AT 12,2 ATTRIBUTE(GREEN, BOLD)

# variable v2_char acquires 7 symbols (from 5th to 11th inclusive) from the
# value of v_max_char

LET v2_char = v_max_char[5,11]
DISPLAY "v2_char      = ",v2_char AT 13,2

# variable v2_char aquires only the 51st symbol of the value of v_max_char
LET v2_char = v_max_char[51]
DISPLAY "v2_char      = ",v2_char AT 14,2
SLEEP 3

DISPLAY "Usage of the Concatenation ( || ) or ( , ) operator:"
      AT 16,2 ATTRIBUTE(GREEN, BOLD)

# The values of v2_char, v2_varchar, and string "That's all right!"
# are concatenated and assigned to variable v_max_char as a single string
LET v2_char      = "This_is_v2_char"
LET v2_varchar   = "This_is_v2_varchar"
LET v_max_char   = v2_char,v2_varchar,"That's all right      !"
DISPLAY "v_max_char = ",v_max_char AT 17,2
```



```
# The values of v2_char, v2_varchar, v1_char and string "That's all right!"  
# are concatenated and assigned to variable v_max_char. As we have one NULL  
# value in the string, the value assigned to v_max_char is NULL too.  
LET v1_char = NULL  
LET v_max_char = v2_char || v2_varchar || v1_char || "That's all right      !"  
DISPLAY "v_max_char = ",v_max_char AT 18,2  
SLEEP 3  
  
DISPLAY "Usage of the CLIPPED operator:" AT 20,2 ATTRIBUTE(GREEN, BOLD)  
  
# The CLIPPED operator caused the variable v2_char and character string  
# "That's all right      " to be displayed without the trailing whitespaces,  
# whereas their actual values remained unchanged  
LET v_max_char = v2_char CLIPPED || v2_varchar || "That's all right      "  
          CLIPPED || "!"  
DISPLAY "v_max_char = ",v_max_char AT 21,2  
  
LET v2_char = NULL  
LET v_max_char = "12345"           "  
LET v2_char = v_max_char CLIPPED -- the CLIPPED operator does not affect  
                           -- the value of the VARCHAR variable  
                           -- v_max_char  
DISPLAY "v2_char = ",v2_char, v_max_char AT 22,2  
SLEEP 3  
  
DISPLAY "Usage of the SPACE operator: " AT 23,2 ATTRIBUTE(GREEN, BOLD)  
  
# The SPACE operator inserts five blank spaces between the values of the  
# variables  
DISPLAY v2_char CLIPPED, 5 SPACE, v2_char CLIPPED AT 24,2  
  
SLEEP 3  
  
END MAIN
```



Numeric Data Types and Arithmetic Expressions

In previous chapters we have discussed two data types: CHAR and VARCHAR. They are the examples of character data types. There are many other data types used by 4GL.

All the data types used by 4GL may be divided into three groups: simple, structured, and large data types. A *simple* data type is a data type that stores a single value. These are the character data types we already know, the numeric data types we will discuss in this chapter, and some other.

A *structured* data type is a data type that can store a structured set of values. The number and the type of the values stored in such variable are specified during the variable declaration.

A *large* data type is a data type containing references to large objects such as text files or binary code. The size of these objects may be up to 2 gigabytes or to a limit determined by your system resources.

In this chapter we will discuss the numeric data types and the arithmetic expressions that can be applied to them by a 4GL program.

Numeric Data Types

Numeric data types are simple data types which store numbers in different representations. When you assign a value to a numeric data type, you need not use quotation marks:

```
LET a = 56
```

If you put quotation marks from both sides of a numeric value, it will be recognized as a character string and will produce a compilation error.

It is necessary to have a notion of *literal numbers* in order to work easily with numeric data types. A *literal number* is a real number in the decimal notation. Integers, fixed-point decimal numbers, and exponential notations are usually represented this way. A literal value cannot include a comma (,) or a blank space, but it can be preceded by a plus or minus sign and include the point (.) sign.

Integer Data Types

The INT, SMALLINT, and BIGINT data types are used to store different data that can be represented by means of whole numbers. Their values are stored as binary integers and have zero scale.

- The *INT (or INTEGER)* values can include numbers ranging from -2,147,483,647 to +2,147,483,647. The INTEGER values take 4 bytes of computer memory. This data type is convenient to perform arithmetic operations, because it doesn't cause rounding errors.
- The SMALLINT data type is similar to the INT data type, but these values are stored as 2-byte binary integers and must be whole numbers that range from -32,767 to +32,767. SMALLINT values are very effective in arithmetic expressions, because they have smaller size and are processed more quickly.
- The BIGINT data type is similar to the INT data type, but the values are stored as 8-byte integers and must be whole numbers that range from -9,223,372,036,854,775,807 to 9,223,372,036,854,775,807



To assign a value to an integer data type, put the numeric expression just after the LET statement:

```
DEFINE a INTEGER  
LET a = 754
```

Remember, that INT, SMALLINT, and BIGINT data types store only whole numbers. If a fractional number is saved to an INT, SMALLINT, or BIGINT variable, its fractional part will be ignored.

The DECIMAL(*p,s*) Data Type

The *DECIMAL* (or *DEC*) data type is intended to store fixed-point and floating-point decimal numbers. You can use it when you want to work with data, such as rates or percentages, which have a fractional part of a specified size and which have to be calculated exactly.

Fixed-Point Numbers

The value of a *DECIMAL*(*p,s*) data type is defined by precision (*p*) and scale (*s*) parameters. The precision part of the *DECIMAL*(*p,s*) value is the total number of digits in the number: before the point and after it. The scale part is the number of decimal places only. There can be up to 32 significant digits in a *DECIMAL* value. The signification of both precision and decimal parts is optional. If you don't specify precision and scale, their default value will be set which is 16 for precision and 2 for scale.

For example, if a variable is declared as *DECIMAL*(15,4), it may have up to 15 significant digits and 4 decimal ones.

When you assign a value to a *DECIMAL*(*p,s*) data type, be attentive: the number of significant digits you or your program set shouldn't exceed the number of significant digits that has been specified when declaring the variable. The symbol that separates the fractional part must be a point (.), not a comma (,):

```
DEFINE my_dec DECIMAL(7,3)  
LET my_dec = 1234.123
```

If the fractional part has more digits than it was specified during the data type declaration, the program will round the value to the allowed number of decimal places.

If you specify the scale larger than the precision, you will receive an error, because the number of decimal places cannot be greater than the total number of digits in a value. The following example will produce an error:

```
DEFINE error_dec DEC(5,9) -- INCORRECT
```

	<p>Note: Remember, that the size of the <i>scale</i> must be smaller than the one of the <i>precision</i>.</p>
--	---

Floating-Point Numbers

You can specify only precision and omit the scale. In such case the variable will be able to store a floating point number the total number of digits in which will be equal to the specified precision.



The MONEY Data Type

The *MONEY* data type is used to store fixed-point numbers which represent currency amounts and have up to 32 significant digits. The 4GL always stores MONEY values as fixed-point decimal numbers. You can specify the precision and scale by using MONEY (*p,s*) format. The default precision and scale are 16 and 2, respectively, and they are applied when no parameters are specified during the data type declaration. When a MONEY variable is displayed, it is preceded by a currency symbol, which is the dollar sign (\$) by default.

All the restrictions concerning assigning values to DECIMAL data type are valid for the MONEY data type. You shouldn't also enter any currency symbols to the program code when you assign a value to a MONEY variable or refer to it:

```
DEFINE my_mon MONEY(7,2)
LET my_mon = 70001.45
DISPLAY my_mon at 5,6
```

The FLOAT/SMALLFLOAT Data Types

The *FLOAT* data type is used to store double-precision floating-point binary numbers which can have up to 16 significant digits. The FLOAT data type may be useful to store and operate with scientific data that don't need to be calculated very precisely. If you want to store precise data, use DECIMAL data type.

The exact representation of a floating-point value may need an infinite number of digits, but the number of digits that can be stored is limited. Therefore, 4GL stores an approximate value, and the least significant digits are equated to zero.

4GL statements can treat FLOAT values as floating-point literals. Floating-point literal has the following structure:



The *SMALLFLOAT* data type is similar to the FLOAT data type, but is used to store single-precision floating-point binary numbers.

There are two ways to assign FLOAT and SMALLFLOAT values – numeric and literal. You can input the numeric value if it doesn't have many digits, or you can use the exponent sign:

```
DEFINE my_float FLOAT,
      my_sffloat SMALLFLOAT

LET my_sffloat = 779.56
LET my_float = 779901.456E-10
```

Arithmetic Operations

An arithmetic operation can be performed on literal numbers, on variables of the above listed numeric data types, and in some cases on variables of the character data types. The numbers stored in CHAR or VARCHAR variables can be operands of arithmetic expressions, only if they consist exclusively of numbers and no other symbols are present. In such case 4GL will be able to treat them as numeric data.



Arithmetic expressions can be used in different statements, and you can use them to assign a value to a variable. Arithmetic expressions return data of numeric data types, use numeric values as operands.

An arithmetic expression must contain at least one arithmetic operator. There are two kinds of arithmetic operators: unary operators and binary operators.

Unary operators are operators that can be applied to one numeric value or to an expression as a whole. 4GL supports two unary arithmetic operators, which are signed by plus (+) and minus (-) symbols. These operators should be put to the left of a value or an expression. They identify whether it has positive (+) or negative (-) meaning. If you don't put any operator before an expression or a value, a default sign (+) will be specified for it.

```
LET b = -341
LET c = +111
LET d = 121
```

Binary operators need at least two values or expressions. 4GL supports six binary operators; their functions and features can be briefly described in a following table:

Operator symbol	Operator Name	Name of Result	Example
+	Addition	Sum	LET a= x+y
-	Subtraction	Difference	LET b = 1000 - y
*	Multiplication	Product	LET c = x * 100
/	Division	Quotient	LET d = x / y
**	Exponentiation	Power	LET e = x**y
MOD	Modulus	Integer reminder	LET f = x MOD y

Before an expression is calculated, its operands are converted to DECIMAL values. If any operand of an expression has a NULL value, the expression will also return the NULL value irrespective of the values of the other operands. A variable can contain a NULL value, if it has not been assigned value after being declared, or if it has been assigned the NULL keyword (e.g. LET a=NULL).

	If you need to put a negative number after the subtraction symbol (-), take the negative number in brackets (e.g. LET a=b - (- c)), otherwise, 4GL will recognize the (--) as a comment indicator
--	---

Each operator has its level of precedence, which determines when the operator will be processed. (+) and (-) have 10 precedence, (*) and (/) have 11 precedence, (***) and (MOD) have 12 precedence. The operators with the highest precedence (here 12 is the highest precedence) are the first to be processed by 4GL, the operators with the lowest precedence are processed last of all. If an expression includes several operators with the same precedence level, 4GL will process them one by one from left to right.

You can change the order of operators processing by using parentheses (()). The operators enclosed in parentheses are processed first of all. Compare:

```
LET a = 12+4/2
```

The variable a will be evaluated to 14.

```
LET b = (12+4)/2
```

The variable a will be evaluated as 8.



The USING Operator

The USING operator specifies display format for values of number or date-time data types. This operator is typically used in the DISPLAY statement, but you can also use it in the LET statement to assign a formatted value to a variable.

Without the USING operator numeric values are displayed using these rules:

- No thousand separators are displayed
- Decimal separators are displayed for all values except INT and SMALLINT
- The trailing currency symbol is displayed for MONEY values

The USING operator must be followed by the pattern which you want to use for the formatting. This pattern must consist only of the specific symbols, described in the table below. Here is the syntax of the USING operator:

```
value USING "pattern"
```

The value can be either a variable of numeric data type, or a literal number. Any other symbol except for the listed below is treated as literal when included into the pattern.

Pattern Symbol	Effect	Example
*	Any blank spaces in the value are replaced with asterisks	123 USING "*****" -> ***123
&	Any blank spaces in the value are replaced with zeros	123 USING "&&&&" -> 00123
#	It does not change any blank positions in the display field. You can use it to specify a maximum width for the space occupied by the value.	123 USING "#####" -> 123
<	Causes the numbers to be left aligned	123 USING "<<<<<" -> 123
,	This character is a literal. USING displays it as a comma (but displays no comma unless there is a number to the left of it).	1230 USING "<,<<<" -> 1,230
.	This character is a literal. USING displays it as a period. You can only have one decimal point (or period) in a number format string.	1230 USING "##.##" -> 12.30
-	This character is a literal. USING displays it as a minus sign when the expression is less than zero and otherwise as a blank. Several minus signs grouped result in one minus sign to the left of the value.	1230 USING "-----" -> -1230
+	This character is a literal. USING displays it as a plus sign when the expression is greater than or equal to zero and as a minus sign when it is less than zero. Several plus signs grouped result in one plus sign to the left of the value.	1230 USING "+++++" -> +1230
\$	The dollar (\$) sign is a placeholder for the front specification of DBMONEY or DBFORMAT environmental variables. When you group several consecutive dollar signs, a single dollar sign floats immediately to the left of the number being printed.	1230 USING "\$\$\$\$\$" -> \$1230



(This literal character is displayed as a left parenthesis before a negative number. Consecutive left parentheses display a single left parenthesis to the left of the number being printed.	
)	For the accounting parenthesis that is used in place of a minus sign to indicate a negative number, one of these characters generally closes a format string that begins with a left parenthesis.	1230 USING "#####"-> (1230)

Example

```
#####
# Numeric data types and operations with them
#####

MAIN

DEFINE
    v_integer      INTEGER,          -- declaring an INTEGER variable
    v_smallint     SMALLINT,         -- declaring an SMALLINTEGER variable which
                                    -- maximum is smaller than that of the INTEGER
    v1_decimal     DECIMAL(5),       -- declaring a floating-point variable with
                                    -- 5 digits total
    v2_decimal     DECIMAL(7,2),     -- declaring a fixed point variable with
                                    -- 7 digits total and 2 DECIMAL places
    v1_money       MONEY(7),        -- declaring a fixed point MONEY variable with
                                    -- 7 digits total
    v2_money       MONEY(7,4)       -- declaring a fixed point MONEY variable with
                                    -- 7 digits total and 4 DECIMAL places

#####
# Part one: assigning values
#####

    DISPLAY "The default values for uninitialized variables are as follows: "
AT 1,2 ATTRIBUTE(GREEN,BOLD)
    DISPLAY "v_integer INTEGER      = ",v_integer,
           "      v2_decimal DECIMAL(7,2) = ",v2_decimal AT 2,2
    DISPLAY "v_smallint SMALLINT     = ",v_smallint,
           "      v1_decimal DECIMAL(5)   = ",v1_decimal  AT 3,2
    DISPLAY "v2_money   MONEY(7)     = ",v2_money,
           "      v2_money   MONEY(7,4)  = ",v2_money   AT 4,2
SLEEP 3
    DISPLAY "Here are the initialized INTEGER variables:" AT 5,2
ATTRIBUTE(GREEN,BOLD)

    LET v_integer = 2147483647 -- maximum value
    LET v_smallint = -32767    -- minimum value
    DISPLAY "v_integer INTEGER      = ",v_integer AT 6,2
    DISPLAY "v_smallint SMALLINT     = ",v_smallint  AT 6,40
SLEEP 3
```



```

        DISPLAY "Here are the initialized floating-point variables:" AT 7,2
ATTRIBUTE(GREEN,BOLD)

        # The result is rounded to 5 digits as the v1_decimal is declared with
        # precision 5
        LET v1_decimal = 1234.5
        DISPLAY "1234.5 DECIMAL(5) = ",v1_decimal AT 8,2
        LET v1_decimal = 1234.567
        DISPLAY "1234.567 DECIMAL(5)= ",v1_decimal AT 8,40
        SLEEP 3
        DISPLAY "Here are the initialized fixed-point variables:" AT 9,2
ATTRIBUTE(GREEN,BOLD)

        # The result is rounded to 7 digits the specified number of which must be
        # decimal places

        LET v2_decimal = 12345           -- two obligatory decimal places
                                         -- are set to 00
        DISPLAY "12345      DECIMAL(7,2)= ",v2_decimal AT 10,2

        LET v2_decimal = 34.5618         -- the number of decimal places is 2,
                                         -- so the value is rounded down
        DISPLAY "34.5678  DECIMAL(7,2)= ",v2_decimal AT 10,35

        LET v2_money = 12.67567          -- the value is rounded up because
                                         -- only 4 decimal places are allowed
        DISPLAY "12.67567 MONEY(7,4) = ",v2_money AT 11,2

        LET v2_money = 129.678           -- zero is added for the missing
                                         -- decimal value
        DISPLAY "129.678   MONEY(7,4) = ",v2_money AT 11,35

        LET v1_money = 1234.67           -- the value fits the declared precision
                                         -- and scale and is not rounded
        DISPLAY "1234.67   MONEY(7)     = ",v1_money AT 12,2

        LET v1_money = 123.4567          -- the value is rounded up because only
                                         -- 2 decimal places are allowed
        DISPLAY "123.4567 MONEY(7)     = ",v1_money AT 12,35

        SLEEP 6
#####
# Part two: operators
#####

        DISPLAY "Unary arithmetic operators + and - :" AT 13,2
ATTRIBUTE(GREEN, BOLD)

        LET v_integer = +12345           -- unary plus sign specifying
                                         -- that the value is positive
        DISPLAY "+v_integer = ",v_integer AT 14,2

        LET v_integer = -12345           -- unary minus sign specifying
                                         -- that the value is negative
        DISPLAY "-v_integer = ", v_integer AT 14,30
    
```



```

LET v_integer = -(-12345)           -- Double unary minus operators
                                         -- result in a plus value
DISPLAY "-(-ingteger) = ", v_integer AT 14,60
SLEEP 3
DISPLAY "Arithmetic operators +, -, *, /, mod, **:" AT 15,2
ATTRIBUTE(GREEN, BOLD)

LET v_integer = 12345      LET v_smallint = 4321
LET v_integer = v_integer + v_smallint + 7761111 -- Addition
DISPLAY "Addition (+)      = ",v_integer AT 16,2
LET v_integer = -7777777
LET v_integer = v_integer - v_integer -(7777777) -- Subtraction
DISPLAY "Subtraction (-)    = ",v_integer AT 16,40
LET v1_decimal = 12345.67e2 * 10          -- Multiplication
DISPLAY "Multiplication (*)   = ",v1_decimal AT 17,2
LET v2_decimal = 12345.67 / -10          -- Division
DISPLAY "Division (/)        = ",v2_decimal AT 17,40
LET v_integer = 8 LET v_smallint = 3
LET v_integer = v_integer MOD v_smallint      -- Modulus
DISPLAY "Modulus (MOD)       = ",v_integer AT 18,2
LET v2_decimal = 2.5 LET v_smallint = -2
LET v2_decimal = v2_decimal ** v_smallint      -- Exponentiation
DISPLAY "Exponentiation (**)" = ",v2_decimal AT 18,40
LET v_integer = 12345 LET v_smallint = NULL
LET v2_decimal = v_integer + v_smallint * 100

# If at least one operand is NULL the whole expression is evaluated to NULL:
DISPLAY "NULL operand: 12345 + NULL * 100 = ",v2_decimal AT 19,2
SLEEP 3

DISPLAY "The USING() operator with numeric data: v2_decimal=-123.456" AT 20,2
ATTRIBUTE(GREEN, BOLD)
LET v2_decimal = -123.456
    # whole numbers or whitespaces:
DISPLAY "#####" = ", v2_decimal USING"#####" AT 21,2
    # whole numbers or zeros:
DISPLAY "&&&&&&" = ", v2_decimal USING"&&&&&&" AT 21,25
    # whole numbers or asterisks:
DISPLAY "*****" = ", v2_decimal USING"*****" AT 21,48
    # displays the dollar sign to the left of the value:
DISPLAY "$$$$$$" = ", v2_decimal USING"$$$$$$" AT 22,2
    # causes numbers to be left aligned:
DISPLAY "|||||<<<<" = ", v2_decimal USING"|||||<<<<" AT 22,25
    # displays minus sign to the left of the value:
DISPLAY "----<<<<" = ", v2_decimal USING"----<<<<" AT 22,48
DISPLAY "The USING() operator with numeric data: v2_decimal=-12345.67" AT 23,2
ATTRIBUTE(GREEN, BOLD)
LET v2_decimal = -12345.67
# only the display of the value is rounded, not the value itself
DISPLAY "----&.&" = ", v2_decimal USING"----&.&" AT 24,2
DISPLAY "<<<<<.=<<" = ", v2_decimal USING"<<<<<.=<<" AT 24,25
# you can use several commas in the USING pattern, but only one dot
DISPLAY "++,++,++.++" = ", -v2_decimal USING"++,++,++.++" AT 24,48
SLEEP 10
END MAIN

```



The Scope of Reference of Variables

The Notion of a Program Module

At the beginning of this tutorial, we have briefly discussed the structure of a 4GL program. We found out that the program consists of statements, which are organized into program blocks, and the program blocks occur in source code files that are called program modules.

So, a *program module* is a source code file containing one or several program blocks. It is possible for a program module not to have the MAIN block, but if it does, it can include only one MAIN block that should be placed at the very beginning of the module (except for some cases described below).

A FUNCTION block is a block which contains a set of commands that comprise a programmer-defined function. A simple function block should begin with the FUNCTION keyword followed by parentheses and end with END FUNCTION keywords. A FUNCTION block may contain the same statements and operators as the MAIN block, and cannot include other program blocks. The differences between the MAIN and the FUNCTION blocks are:

- Any program must include one and only MAIN program block, whereas there can be no or any number of FUNCTION blocks.
- The MAIN block is executed only once during the program operation. When the last statement of the MAIN program block is executed, the program is finished. A FUNCTION block, in its turn, can be called and executed unlimited number of times.
- The MAIN block is executed automatically and obligatory, when the program is run. The FUNCTION block must be called during the program execution, if needed, and may be not executed at all.

A program module that includes only FUNCTION blocks and no MAIN block is called a *function library*.

A Multi-Module Program

A program can consist of more than one module - of more than one source code file. In such case at least one of its modules must contain the MAIN program block. The number of FUNCTION blocks and the modules in which they are located is not limited. When you compile your application, Lycia automatically links the files together. If any file cannot be compiled, the compilation of the whole program cannot be completed. For more information about the program compilation, see "Lycia Developer's Guide" and "Lycia Introduction" documents

Types of Variables with Regard to Their Scope

We have discussed variable declaration within the MAIN block in previous chapters. This is not the only way to declare a variable. 4GL supports three methods of variable declaration depending on where you want to use the declared variable.



Local Variables

A variable can be declared and used within a program block (MAIN or FUNCTION). Such variable is called a *local* variable. These variables can be referred to only by the statements that are located within the same program block. Until now we used only local variables in the examples within this manual.

There are also variables whose scope of reference is even more limited than the one of the local variables. You can specify the variables that will be used only within a part of a program block and won't be available to the statements outside this part.

Such part of a source code is called a *logical sub-block*. The logical sub-block starts with the BEGIN and ends with the END keywords and needs no identifier. It can refer to the local variables of the program block where it is located and can contain declarations of the variables available only within this block, so-called *spot variables*:

```
MAIN

    DEFINE var1 integer
    LET var1 = 5

    BEGIN -- the beginning of the logical sub-block
        DEFINE var2 integer -- a spot variable declaration
        LET var2 = 10
        DISPLAY "var1=", var1
        DISPLAY "var2=", var2
    END -- the end of the logical sub-block
END MAIN
```

If you try to reference a spot variable from outside its logical sub-block, a compile-time error will occur.

Module Variables

Sometimes, it is necessary for a variable to be available to all or several program blocks of one module. To declare such variable, you must put the declaration clause *before* the MAIN program block:

```
DEFINE mod_var1 integer,
      mod_var2 character(20)

MAIN

...
END MAIN
```



The variables, available to all the program blocks of a module, are called *module* variables. If a module does not contain the MAIN block, the module variables declaration should take place before the first FUNCTION block.

Global Variables

The third type of 4GL variables are *global* variables. Global variables are the variables that can be referenced by any module or program block of the program. Global variables are declared with the help of the GLOBALS statement. The GLOBALS statement must include at least one DEFINE statement. It has the following syntax:

```
GLOBALS
  DECLARE
    <variables declared>

  END GLOBALS
```

The GLOBALS statement can be located in a separate file, which is called *GLOBALS file*, or within any other module.

If it is located within a program module that contains other program blocks, it should be placed before all of them, even before the MAIN block and before the DEFINE statement preceding the MAIN block, if there is one:

```
GLOBALS

  DEFINE mod_var1 INTEGER,
        mod_var2 CHARACTER(20)

  END GLOBALS

  DEFINE a CHAR(20)

  MAIN
  ...
  END MAIN
```

The GLOBAL variables declared this way may be used only by the program blocks located within the same module. It means that such GLOBAL variables are similar to the MODULE ones. Use this method only if you have a program consisting of one program module.

If you want your GLOBAL variables to be available for all the program blocks and modules, you should put the GLOBAL statement into a separate file (the GLOBALS file). No other program blocks should be present in such GLOBALS file.

To refer to global variables located in a GLOBALS file, you should put the GLOBALS “*filename*” statement at the very beginning of each module that is to refer to the global variables. The “*filename*” must include the name of the GLOBALS file, its extension, and its path in case it is not located in the same folder as the program module.

Here is an example of the usage of GLOBAL variables, stored in a GLOBALS file. The GLOBALS file *my_globals.4gl* has the next source code:



```
GLOBALS  
  
DEFINE my_age INTEGER  
  
first_name, last_name CHAR(20)  
  
END GLOBALS
```

You shouldn't enter any program blocks after the END GLOBALS statement if you want the GLOBALS file to be available to other modules.

Now you can refer to these variables in your source code, but don't forget to specify the GLOBALS file:

```
GLOBALS "my_globals.4GL"  
  
MAIN  
  
LET my_age= 20  
  
LET first_name= "Jack"  
  
LET last_name = "Jones"  
  
END MAIN
```

These LET statements will refer to the values, stored in *my_globals.4gl* file.

You can link several GLOBAL files to one program module. For example, the following source code is possible:

```
GLOBALS "my_globals.4GL"  
  
GLOBALS "my_globals2.4GL"  
  
GLOBALS "additional.4GL"  
  
MAIN  
...  
...  
END MAIN
```

Correlation Between Variables of Different Types

When you declare different types of variables, you should remember a set of rules to avoid errors and invalid values.



- You cannot refer to a variable local to one program block from another program block - it will not be visible
- The names of local variables located in different program blocks and modules may match. This won't affect the program execution as these variables have non-overlapping scopes of reference.
- If you declare the same names for MODULE and GLOBAL variables, the MODULE one will be referenced by the statements used in its module and the global one will be ignored in this module, but used in all others. The same is true for local variables with regard to global and module ones.
- You must put the GLOBALS "*filename*" statement exactly at the very beginning of the program module.
- 4GL checks the global and module variables names against the built-in variables names in order to prevent conflicts between them. If the name of your global or module variable matches one of the built-in variables names, the 4GL will produce a compile-time error. However, such check isn't performed for local variables, so if a local variable name matches a built-in variable name, no compile-time error will occur, but the program may return unexpected values. Here is the list of the built-in 4GL variables: *int_flag*, *quit_flag*, *SQLAWARN*, *SQLCA record*, *SQLCODE*, *SQLERRD*, *SQLERRM*, *SQLERRP*, *status*.

The FUNCTION Program Block.

As we have already mentioned, a *function* is a named set of statements that causes a program perform a number of actions. Functions are very convenient when you need to use the same set of operations for several times. We will discuss functions in details later in this manual.

The FUNCTION program block begins and ends with the FUNCTION and END FUNCTION keywords, respectively. The FUNCTION block cannot be located within the MAIN or another FUNCTION program block and cannot contain any of them. The general view of the FUNCTION program block is as follows:

```
FUNCTION function_name( )  
    ...  
    ...  
    ...  
}  
    function body  
END FUNCTION
```

The *function_name* is the identifier of the function using which you can call this function from anywhere within your program. The restrictions on function names are similar to the restrictions on variable identifiers. The name of the function must be unique among other functions within a program. The parentheses should immediately follow the function name. In this chapter we will discuss only functions in which the parentheses are empty.

The *function body* can contain any executable statements including the DEFINE statement; here you specify the actions which should be performed when a function is called.

Here is a very simple example of a function:



```
FUNCTION display_string ()  
    DISPLAY "My first function" at 3,4  
END FUNCTION
```

This function must display the string "*My first function*" to the screen, but to do it, the function must be invoked, or called from outside this FUNCTION block. Usually a function is called from the MAIN block, but it can also be called from another function, provided that this second function is called from the MAIN block itself. You can nest any number of callings in this fashion, but be sure that calling starts from the MAIN block.

To call the function, use the CALL statement. The simplest case of the CALL statement is as follows:

```
CALL function_name()
```

So, to call the function *display_string()*, you should include the CALL statement into the MAIN block. Here is an example where the function and the MAIN block are located in one module:

```
MAIN  
CALL display_string() --the function is invoked  
END MAIN  
  
FUNCTION display_string ()  
    DISPLAY "My first function" at 3,4  
END FUNCTION
```

The order in which the FUNCTION blocks are located in the program code, does not affect the order of their execution. The order of execution depends only on the order of the functions invocation.

Here is a simple example of invocation of a function which is located in the same module. In this example local and module variables are used.

```
DEFINE  
    a integer  
  
MAIN  
  
    DEFINE b integer  
    CALL get_a()  
    LET b = a*2  
  
    DISPLAY "a=", a, " b=", b  
  
END MAIN  
  
FUNCTION get_a()
```



```
LET a=3
```

```
END FUNCTION
```

The function *get_a()* is called by the CALL statement located in the MAIN program block. The variable *a* is a module variable, its value is assigned in the *get_a()* function and used in the MAIN block. The variable *b* is a local variable and is used in the MAIN block only.

A Function Library

The functions created within one 4GL program can be called not only from the program module where they are located, but also from other program modules. Sometimes it is convenient to put all the functions you need to a function library module and then to refer to these functions from the main module or from other function libraries.

When you call a function that is located in another program module, it is often necessary that the variables used by this function are of the global scope (there are ways allowing to use local ones, but they are discussed in the next chapter). This will let the modules which call the functions and the functions use the same variables.

Here is an example of creating a function and calling it from a different module.

First, you should create a GLOBALS file "my_globals.4gl" which contains all the variables that will be needed for both functions and the program blocks that call them

```
GLOBALS

    DEFINE first_name, last_name, city, street CHAR (20)

        age, house_nr INTEGER

    END GLOBALS
```

Now let's create a function library with three FUNCTION blocks in a new file, and its source code will be:

```
GLOBALS "my_globals.4gl"

FUNCTION get_name() -- first block

    LET first_name = "Jack"

    LET last_name = "Jones"

END FUNCTION

FUNCTION get_age() -- second block

    LET age=35

END FUNCTION

FUNCTION get_addr() -- third block
```



```
LET city="London"  
LET street="Grace str."  
LET house = "34"  
END FUNCTION
```

The `get_name()` function assigns values to the `first_name` and the `last_name` variables. The `get_age()` function assigns the value for the `age` variable. The `get_addr()` function assigns values for the `city`, the `street`, and the `house` variables.

Now, we can call these functions from the module where the MAIN block is located and display the personal data of a person to the screen:

```
GLOBALS "my_globals.4gl"  
  
MAIN  
  
CALL get_name()  
CALL get_age()  
CALL get_addr()  
  
DISPLAY "His name is ", first_name CLIPPED, " ", last_name CLIPPED, " he is  
", age, " years old." AT 1,2  
DISPLAY "He lives in ", city CLIPPED, " ", street CLIPPED,  
" House #", house AT 2,2  
  
END MAIN
```

The MAIN program block calls all the functions from the function library and displays the global variables values, assigned in these functions.

	Note: To call a function, located in another module, you don't have use any additional code, the CALL statement is enough.
--	---

Example

This application consists of several modules and illustrates the way variables are referenced in a program depending on the place where they are declared.

The GLOBALS File

```
#####
# This is the file "global_block.4gl" which contains global variables.
# These variables are visible in all modules and blocks of this program.
# No other blocks are allowed here.
#####
```



GLOBALS

```
# The GLOBALS block must include the DEFINE statement:  
  DEFINE  
    g_char, OUR_COMPANY CHAR (80)  
      -- Here we declare two global variables,  
      -- they are of the same data type  
  
  END GLOBALS  
  
#####  
# This is another GLOBALS file "global_block1.4gl" where one global variable  
# is declared. It has been included in the program to demonstrate that  
# several GLOBALS files can be linked to one module.  
#####  
  GLOBAL  
  
  DEFINE  
  
    matching_var CHAR(80) -- Another global variable  
  
  END GLOBALS
```

The MAIN Program Module

Here is the program module containing the MAIN block:

```
#####  
# This is the main module of our program called "main_block.4gl", it contains  
# the only MAIN...END MAIN block we have. It can also contain any number  
# of function blocks after the MAIN block, our file has one such block.  
#####  
  
# Here we link the files containing the global variables  
  GLOBALS "global_block.4gl"  
  GLOBALS "global_block1.4gl"  
  DEFINE  
    m_char CHAR(80)      -- The module variable m_char is visible in both  
                          -- the MAIN block and the FUNCTION block within this  
                          -- module.  
  
  MAIN  
  
  DEFINE  
    l_char CHAR(80)      -- The declaration of the local variable l_char,  
                          -- visible only within the MAIN program block  
  
  # initializing global variables  
    LET OUR_COMPANY      = "QUERIX"  
    LET g_char            = "global"  
    LET matching_var     = "glob_scope"  
  
  # initializing module variable (module main_block.4gl)  
    LET m_char           = "m_main_block"
```



```
# initializing local variable (local to MAIN block)
LET l_char          = "local_main"

DISPLAY "The values assigned to the variables in the MAIN block: are:"
AT 1,2 ATTRIBUTE (GREEN, BOLD)
DISPLAY "The global variable OUR_COMPANY = ",OUR_COMPANY CLIPPED AT 2,2
DISPLAY "The global variable g_char = ",g_char CLIPPED AT 3,2
DISPLAY "The global variable matching_var = ",matching_var CLIPPED AT 4,2
ATTRIBUTE (BOLD)
DISPLAY "The module variable of main_block.4gl module m_char = ",m_char AT 5,2
ATTRIBUTE (YELLOW,BOLD)
DISPLAY "The local variable of the MAIN program block l_char = ",l_char AT 6,2
ATTRIBUTE (YELLOW,BOLD)

BEGIN           -- The sub-block with a spot variable declaration begins
DEFINE sp_var STRING -- This is a spot variable

LET sp_var = "This is a spot variable which is not visible outside",
        "its sub-block"
DISPLAY sp_var AT 7,2 ATTRIBUTE (BLUE, BOLD)
END             -- this is the end of the sub-block

# If you uncomment the next line, the program will fail to compile because
# sp_var is a spot variable and it cannot be seen by the other program
# blocks, including the MAIN program block

-- DISPLAY sp_var AT 8,2

# this is a simple function call which invokes func1() located in this
# module:
CALL func1()

SLEEP 40
END MAIN

#####
# The beginning of the FUNCTION program block
#####

FUNCTION func1() -- declaring the name of the function

# Here begins the function body
DEFINE
    l_char CHAR(80), -- This variable is visible only within the
                      -- FUNCTION func1() program block.
                      -- Pay attention that it has the same name as the local
                      -- variable of the MAIN block, but as both are local, the
                      -- scopes of their reference do not intersect.

    matching_var CHAR(80) -- We declare a local variable with the same
                          -- name as the global variable. If this
                          -- variable hadn't been declared, the
                          -- corresponding global variable would have
                          -- been used instead.
```



```
# The LET statement is used to initialize the local variables.  
# The MAIN local variable l_char and matching_var global variables do not  
# have an effect on the value of these variables.  
# These variables would have NULL values if not initialized.  
  
LET l_char      = "local_func1"  
LET matching_var = "local_scope"  
  
DISPLAY "The variables values after processing the FUNCTION func1()",  
" program block:" AT 9,2 ATTRIBUTE (GREEN, BOLD)  
# lines of the text as one and the same one  
# These variables retain the values assigned in the MAIN block:  
DISPLAY "The global variable OUR_COMPANY = ",OUR_COMPANY CLIPPED AT 10,2  
DISPLAY "The global variable g_char = ",g_char CLIPPED AT 11,2  
DISPLAY "The module variable of main_block.4gl module m_char = ",m_char CLIPPED  
AT 12,2  
  
# These variables have new values:  
DISPLAY "The local variable of the FUNCTION fun1() program block ",  
"matching_var = ",matching_var CLIPPED AT 13,2 ATTRIBUTE (YELLOW,BOLD)  
DISPLAY "The local variable of the FUNCTION fun1() program block ",  
"l_char=",l_char CLIPPED AT 14,2 ATTRIBUTE (YELLOW,BOLD)  
  
# Here we assign new value for the global variable which has already been  
# initialized once the new value replaces the old one.  
LET g_char = "global_changed"  
  
# this is a simple function call which invokes func2() located in another  
# module - "func_block.4gl".  
CALL func2()  
  
END FUNCTION
```

The Function Library

Then here is a function library - the module which contains only FUNCTION blocks and no MAIN block:

```
#####
# This module is called "func_block.4gl" and it is a function library.
#####

GLOBALS "global_block1.4gl" -- Here we link the files containing the global
-- variables
GLOBALS "global_block.4gl"

DEFINE
  m_char CHAR(80)    -- The module variable m_char is visible only in this
                      -- file and is NOT visible in the "main_block.4gl"
                      -- module.
                      -- These modules have module variables with the same
```



```
-- names.

#####
# The beginning of the function program block.
#####
FUNCTION func2() -- here is the name of the function

DEFINE
    l_char CHAR(80) -- this local variable is visible only within the
                      -- "func2()" function block
                      -- It has the same name as the local variables in other
                      -- modules, but they do not conflict due to the different
                      -- scope of reference.

# Here we use LET to initialize the local and module variables.
# Though they have the same names as previously initialized variables,
# they are completely different variables and do not inherit the values
# assigned to their counterparts with another scope of reference.

LET m_char = "m_func_block"
LET l_char = "local_func2"

DISPLAY "Here are the values of the variables after processing the ",
        "FUNCTION func2() program block:" AT 17,2 ATTRIBUTE (BOLD, GREEN)

DISPLAY "The global variable OUR_COMPANY = ",OUR_COMPANY CLIPPED AT 18,2
          -- OUR_COMPANY value was assigned in main block
DISPLAY "The global variable g_char = ",g_char CLIPPED AT 19,2
          -- g_char value was changed in function func1()
DISPLAY "The global variable matching_var = ",matching_var CLIPPED
          AT 20,2 ATTRIBUTE (BOLD)
          -- the global matching_var variable retains the value
          -- assigned in the MAIN block

DISPLAY "The module variable of func_block.4gl module m_char = ",
        m_char CLIPPED AT 21,2 ATTRIBUTE (YELLOW,BOLD)
          -- m_char value was assigned in this function

DISPLAY "The local variable of the FUNCTION fun2() l_char = ", l_char CLIPPED
          AT 22,2 ATTRIBUTE (YELLOW,BOLD)
          -- l_char value was assigned in this function

END FUNCTION
```



Functions

In the previous chapter we have discussed the basic idea of a function. We found out that a function is a named set of statements that may be called and executed during the program operation.

4GL differentiates between programmer-defined functions and built-in functions.

- *Built-in functions* are stored in the 4GL syntax and need not to be defined. The actions which they instruct a program to perform are fixed and cannot be changed.
- *Programmer-defined functions* are created by a programmer. Therefore they are more flexible, a programmer can specify any set of actions for a program to perform. On the other hand, they need to be declared. The function declaration contains the actions to be performed by the function, as it is not built-in and 4GL does not know what you want it to do without the declaration. A programmer defined function is created by means of the FUNCTION statement and is located in the FUNCTION program block.

The Function Definition and Declaration

We have already discussed a subset of programmer defined functions which do not return any values and do not accept any arguments. These belong to the simplest function type. Now we will discuss the functions which are more flexible, because they can accept arguments and return values.

A FUNCTION declaration requires specification of the function name *identifier* and optional formal arguments. We have discussed the restrictions influencing the function identifier in the previous chapter.

The statements between the function declaration and the END FUNCTION keywords comprise the *function body*.

	<p>Note: When you develop your application, you can specify a function with an empty function body in order to test other parts of your program:</p> <pre>FUNCTION my_func() END FUNCTION</pre> <p>Such function doesn't perform any actions, but it will prevent the program from a compile-time error.</p>
--	--

Formal Function Arguments

The *formal function arguments* are identifiers (variables) which receive values transferred to the function from the calling routine when its execution starts. These values are transferred by the actual function arguments specified in the CALL statement which is described below. The formal arguments should be listed



in parentheses following the function name, they must be unique within the current argument list. The arguments must be separated by commas.

If you specify no arguments, you must indicate an empty argument enclosed in the parentheses (for ex., FUNCTION my_funct ()).

The function arguments may be represented by variables of local, module, and global scope. The local variables should be declared within the function they are used by, and the declaration clause should follow the argument list immediately. If a global or a local variable name matches the function local variable name, the latter will be used by the function statements, and the former ones will be ignored.

```
FUNCTION function_name (arg1, arg2)

DEFINE
    arg1, arg2, arg3 INT

LET arg3=arg1+arg2

DISPLAY arg3

END FUNCTION
```

The main purpose of using arguments is that the variables used as arguments receive their values from outside the function - from the calling routine. Thus in the example above we do not assign any values to *arg1* and *arg2* because their values are passed to the function, so we can perform an arithmetic operation on them.

The CALL Statement

Any function cannot be executed without having been specially invoked. The CALL statement is used to invoke a function. We have already touched upon this statement, but here it will be discussed in more details.

The CALL statement can be used to call a programmer-defined 4GL function, a built-in 4GL function, or a C function. When the CALL function is executed, the program control is passed to the specified FUNCTION program block, and the program starts to execute the statements, specified in the function body.

The program block which contains the CALL statement is called a *calling routine*.

Actual Function Arguments

A useful feature of the CALL statement is its ability to pass some values as *actual arguments* to the function. It means, that some values used by function may be specified from outside this function. The actual arguments are to be given in parentheses following the function name specified in the CALL statement.

If the argument list contains several arguments, they must be separated with commas.

```
CALL my_func (5,6)
```

The actual arguments can be either variables of any scope of reference, 4GL expressions, or literal values, such as a character string, a literal date, or a number. This is one of the major differences between the



formal function arguments and actual function arguments, because the formal arguments can be represented only by variables.

Only the values can be transferred to a function, the variables used to specify them are not transferred. Thus the values of the variables used as actual arguments are assigned to the variables used as formal arguments. This imposes some restrictions on the both argument lists:

- The number of formal arguments must be equal to or more than the number of actual arguments, sent to the function by the CALL statement.
- The arguments must also match in order and be of the same or of compatible data types.



Note: Data types are called *compatible* when 4GL provides automatic data type conversion for their values. The success of the conversion often depends on the value length, precision and scale, and other features of a converted value. The description of compatible data types and the rules of conversion are given [Invoking a Function Without the CALL Statement](#)

If a function returns exactly one value, you can use it as an operand of an expression. In such case the CALL statement is not needed. When using a function as an operand you can still pass a number of arguments to it. The arguments must be placed in parentheses following the function name. If there are no arguments, the empty parentheses should appear.

```
MAIN

DEFINE i INT

#the function is called implicitly
and the value returned by it is
#used as a part of an arithmetic
expression
LET i = my_func()*2

DISPLAY i AT 2,2      -- displays "30"

END MAIN

FUNCTION my_func()

DEFINE a_f INT

LET a_f = 15

RETURN a_f -- the value of the
variable a_f is to
-- be sent to the
calling routine.

END FUNCTION
```

Compatible Data Types.

The order in which the actual arguments are listed determines the order in which their values are passed to the formal function arguments.

```
MAIN

DEFINE a,b INT

LET a=4
```



```
CALL func_2(a+1,6) -- the values transferred to the function are
-- 5 and 6

END MAIN

FUNCTION func_2(af,bf) -- the values are transferred as follows:
-- af=5, bf=6

DEFINE

    af, bf INT,
    cf INT

    LET cf=af+bf

    DISPLAY "cf= ", cf clipped AT 2,2

END FUNCTION
```

As the result of the program execution, the following string will appear in the screen:

cf=11



Note: If you use a variable with a NULL value as an argument in a CALL statement, the result may be unpredictable.

It is possible that you will have to pass a sequence of two or more values to one function formal argument. You have to unite (or to concatenate) them before such procedure can be executed. The concatenation is performed if you put a double-pipe symbol (||) between the two values. The two values concatenated with a double-pipe symbol will be passed as a single value to the corresponding formal argument:

```
MAIN

DEFINE f_name,mid_name CHAR(5),
       l_name CHAR (10)
LET f_name = "Jane "
LET mid_name= "Alice"
LET l_name="Greenberg"

CALL fullname(f_name||mid_name,l_name)
END MAIN

FUNCTION fullname(a,b)
```



```
DEFINE a, b CHAR(10)
DISPLAY "The full name is: ", a, " ", b
END FUNCTION
```

In this example, the variable *a* gets the value made of the values of the variables *f_name* and *L_name*, and the variable *b* gets the value of the variable *_name*. As the result of the program execution, the following line will be displayed to the screen:

```
The full name is: Jane Alice Greenberg
```



Not: If you use a double-pipe symbol within an argument list, the elements concatenated by it are treated as a single argument, so be attentive and carefully check the number of the actual and formal arguments.

A function can be called from within itself. In such case it will become a recursive function. However, you should not try it so far, because it will cause an infinite loop unless you place the recursive function call into a conditional statement. These types of statements are discussed later in this manual and there you will see an example of a recursive function call.

The RETURN Statement and the RETURNING Clause

A function can return some values to the calling routine in order for the program to use it during the further execution. If you want the function to perform such operation, you must use the RETURN statement.

The RETURN clause is optional, it consists of the RETURN statement and a variable list, and is situated at the very bottom of the function body, just before the END FUNCTION keywords:

```
FUNCTION my_func()
...
RETURN a, b
END FUNCTION
```

The RETURN statement can be used only within the FUNCTION block. If you try to include it to another program block, a compile-time error will occur.

The RETURN clause may be empty. In such case it does not return any values, it just returns the program control to the calling routine.

```
MAIN
...
CALL my_func()
END MAIN
```



```
FUNCTION my_func()  
    ...  
    RETURN  
END FUNCTION
```

If the RETURN statement is followed by a list of values, these values will be transferred to the calling routine after the function is executed. The values list must follow the RETURN keyword immediately and the values must be separated with commas.

In order for the RETURN statement to operate correctly, you must add the RETURNING clause after the function name used in the CALL statement. The RETURNING clause must include the variables that will get the returned values. These values are transferred to the calling routine in the same way the arguments are transferred to a function. The values of the variables or the actual values in the RETURN statement are assigned to the variables of the RETURNING clause of the CALL statement used to invoke the function. These variables should match those in the RETURN clause in number, order, and data types (or be of compatible data types).

```
MAIN  
  
DEFINE a, b INT  
  
CALL my_func() RETURNING a,b --indicates the variables  
                           -- to receive the returned values  
  
DISPLAY "a=", a clipped, 5 SPACE, "b=", b clipped  
  
END MAIN  
  
FUNCTION my_func()  
  
DEFINE a_f, b_f INT  
  
LET a_f = 15  
LET b_f = 30  
  
RETURN a_f, b_f --values of a_f and b_f are assigned to variables a  
                  --and b accordingly  
  
END FUNCTION
```

The program above calls a function that returns two integer values to the calling routine. The function *my_func()* assigns values to two local variables (*a_f* and *b_f*) and then transfers these values to the local variables of the MAIN block (*a* and *b*) of the MAIN program block. As a result, the following string will be displayed to the screen:



a=15 b=30

You should remember that the order in which variable values are transferred corresponds to the order, in which the variable values are got, so be attentive when you specify variable lists.

Invoking a Function Without the CALL Statement

If a function returns exactly one value, you can use it as an operand of an expression. In such case the CALL statement is not needed. When using a function as an operand you can still pass a number of arguments to it. The arguments must be placed in parentheses following the function name. If there are no arguments, the empty parentheses should appear.

```
MAIN
  DEFINE i INT
    #the function is called implicitly and the value returned by it is
    #used as a part of an arithmetic expression
    LET i = my_func()*2
    DISPLAY i AT 2,2      -- displays "30"
  END MAIN

  FUNCTION my_func()
    DEFINE a_f INT
    LET a_f = 15
    RETURN a_f  -- the value of the variable a_f is to
                 -- be sent to the calling routine.
  END FUNCTION
```

Compatible Data Types

Sometimes data type conversion may be useful when you need to process, to store, or to compare some values. The ability to convert data types correctly can sometimes make it possible to make the source code shorter and easier to read.

Only compatible data types can be successfully converted into one another. It is very important for a 4GL program developer to have an idea of compatible data types, and this knowledge will prevent them from writing a source code that produces incorrect values or errors.

The compatible data types are the types we have already discussed (i.e. simple data types). Most of them can be converted into one another, but there is a list of general rules and restrictions:



- Any data type can be converted into CHAR, VARCHAR and STRING data types, and they may be converted into any other data type, if the initial value matches the format of the necessary resulting value (e.g. a CHAR value can be converted into INT, if it consists exclusively of digits). If the result of the conversion of a value into the CHAR, VARCHAR OR STRING data type is longer than it is permitted, the excessive characters of the string will be truncated from the right.
- The parameters (size, precision, scale, etc) of the initial value should correspond to the ones of the resulting value, otherwise, the value will be changed during the conversion.
- If a data type that has a fractional part is converted into a data type that doesn't have one, the fractional part will be ignored.
- If the converted value has more significant digits than it is allowed by the resulting data type, the low-order digits will be ignored and discarded.

Built-In Functions Used with Character Values

As it was mentioned above, 4GL has a list of built-in functions that may be called during the program execution. In this chapter we will discuss some of these functions.

Functions Managing ASCII Code

Each symbol recognized by 4GL has its own code (ASCII standard code), by which it is referred by the program. Thus, the code number of the lower-case *a* is 97, the code number of the upper-case *A* is 65, the code number of the digit 7 is 55. The punctuation marks and other symbols also have their codes.

Converting Letters into ASCII Code

The *ord()* function is used to evaluate the first character of its argument and to return the integer value corresponding to the code number of this character.

```
DEFINE my_ord INT  
  
LET my_ord = ord("My ord() function") --returns 77 - the code of M  
-- character
```

Remember, that the *ord()* function is case sensitive and will return different values for the same letters put in the upper and lower case. For example, if you replace the letter "M" in the example by the letter "m", the value returned will be 109.

The argument of the *ord()* function may be of any length, but only the first character will be evaluated, unless the argument is represented by a variable, and the number of the ordinal number of the necessary character is given in square brackets:

```
DEFINE a CHAR(5), my_ord INT  
  
LET a = "Hallo"  
  
LET my_ord = ORD(a[5])
```



The variable *my_ord* will get the value 111 which corresponds to the lower-case letter *o*.

Converting ASCII Code to Letters

The ASCII function is a function, reverse to the ORD() function. The function evaluates its integer argument and returns a character, whose code corresponds to the given number. The ASCII argument can range from 0 to 127 – this is the number of characters, recognized by 4GL. If the ASCII argument exceeds the given limits, a run-time error will occur.

The following example will display "Hello":

```
DISPLAY ASCII 72, ASCII 101, ASCII 108, ASCII 108, ASCII 111 AT 2,2
```

Changing Letter Case

The *upshift()* and *downshift()* functions are used to convert their character-string arguments. The *upshift()* function returns a string with the lower-case letters converted into upper-case letters. The *downshift()* function returns a string with the upper-case letters converted into lower-case letters. The maximum size for the both functions arguments is 32,766 bytes.

Both functions are applied to character values. For example, they may be used to prevent similar strings from being recognized as different ones, if their meanings are equal (e.g., "Monday" and "MONDAY", "USA" and "usa", etc.).

If you use the *upshift()* function, the upper-case letters are not changed. If you use the *downshift()* function, the lower-case letters are not changed. In both cases of non-alphabetic characters are also not changed.

```
LET country = Australia
LET down_sh = DOWNSHIFT(country) --returns "australia"
LET up_sh = UPSHIFT(country)      --returns "AUSTRALIA"
DISPLAY down_sh, up_sh
```

The source code given above converts the value of the variable *country* into a lower-case and upper-case character strings and then displays them.

The success of uppercase letters conversion into lower-case ones and vice versa depends on the locale files. If they contain such information, the conversion will be executed correctly, if not - no conversion will be performed. Non-ENGLISH characters are mostly not affected by *upshift()* and *downshift()* functions, especially, in multibyte locales.

Getting the Length of a String



The *length()* function returns an integer value that corresponds to the length of its character string argument.

There are several possible reasons to use the *length()* function. Thus, the program can check whether the user has entered a valid value or a filename. If the *length()* function argument is a *null* string, the function will return zero.

```
DISPLAY LENGTH("test") AT 2,2 -- will display 4
```

	Note: The LENGTH function ignores automatically added blank spaces, so that they do not affect the value returned by this function.
--	--

Truncating a String

The *trim()* function removes any blank spaces from the beginning and from the end of a character string or a value of a character data type variable, even if they were specified by the user or by the programmer (unlike the CLIPPED operator which removes only the blank spaces added automatically to fit the data type size).

If the argument of the *trim* function includes several words with blank spaces between them, these blank spaces are not deleted.

The following source code line:

```
DISPLAY trim (" Hello ") , trim(" ", World) , "!"
```

Will produce the following result:

```
Hello, World!
```

You should keep in mind, that the *trim()* function can have only one argument. If you want to trim several values, you have to use the *trim()* function for several times.

The *trim()* function can be applied only to character strings or to character variables.

The Function Returning a Random Number

The *fgl_random()* function returns a random decimal number between and including the two numbers specified as its arguments.

The first argument specifies the minimum possible number; the second specifies the maximum one.

The following example demonstrates how you can make the program generate a random number ranging from -100 to 100:

```
LET rand_val = fgl_random(-100,100)
```

The value returned by the *fgl_random()* function is a decimal number. If you assign it to an integer variable, the value will be rounded.



Verifying the Arguments Passed to the Application

When launching a 4GL application, you can pass a number of arguments to it. It is either done by adding the arguments to the command, if the application is run using the command line, or by specifying them in the program properties, if you use the Studio. During the execution of the application you can verify whether it has been run with arguments and what they were using the built-in functions.

Retrieving Individual Arguments

The *arg_val()* function gives the possibility to return one or several arguments specified in the command line that was used to run the program. The arguments you want to be passed to the program must be listed in the command line just after the program name and they mustn't be separated with commas. The example below sends two arguments to the program

```
C:\\\\...\\\\output>qrun functions argument1 argument2
```

For more information about launching programs see the "Lycia Developer's Guide".

The *arg_val()* function is the part of the source code that makes the arguments transfer possible. The function can be called from any program block, and may pass values to the variables specified in the MAIN block, which may prove useful sometimes. The *arg_val()* function makes it possible to retrieve individual arguments when the program is being executed.

The syntax of the function invocation is as follows:

```
arg_val(ordinal)
```

Where *ordinal* corresponds to a non-negative integer expression that indicates the ordinal number of the argument listed in the command line. If the *ordinal* value is 0, the *arg_val()* function will return the name of the program.

The following part of the source code demonstrates how the *arg_val()* function is used to chose one of the arguments listed in the command line (Provided that the program was invoked as: *qrun functions 456 argument_1*):

```
DEFINE  
  
    v1_char CHAR(60)  
  
    ...  
  
    LET v1_char = ARG_VAL(2) -- returns argument_1  
  
    DISPLAY "Arg_val(2) = ",v1_char AT 17,2
```



Note: When you enter the executable file name and the arguments to the command line, make sure that you are doing it from the folder which contains the file.



Retrieving the Number of Passed Arguments

The *num_args()* function doesn't need arguments and is used to return an integer value that shows how many arguments have been passed to the program from the command line. The function cannot be used in order to retrieve individual arguments.

If you enter the following string to the command line

```
qrun Functions 456 argument_1
```

the *num_args()* function will return the value 2, because two arguments are passed to the program.

Verifying Application Launch Mode

The *fgl_fglgui()* function is used to indicate whether the program is run in a graphical or in character mode. The result is obtained on the basis of the value of the GUI environment variable, which is *1* if the application is run in a GUI mode and is *0* if the application is run in the character mode. The function needs no arguments. Below is given an example of the function application:

```
DEFINE a INTEGER
LET a = fgl_fglgui()

DISPLAY a
```

If the user runs the program in the character mode, the variable *a* gets the value *0*, if the program is performed in a GUI mode, the variable *a* gets the value *1*.

The function may prove useful, for example, if your program output looks different in character and in GUI mode, so you can specify the way the program displays the information.

The *fgl_getuitype()* function is used to return a character string which contains the information about the type of the current user interface.

The function can return one of the four values given in the table below.

Value	Meaning
CHAR	A character-based terminal
WTK	A Windows front end
JAVA	A Java front end
HTML	An HTML front end

The syntax and the application of the function is similar to those of the *fgl_fglgui()* function:

```
DEFINE a VARCHAR
LET a = fgl_getuitype()
```



```
DISPLAY "The user interface type is ", a
```

If you run the program in character mode, variable *a* will get the value *CHAR*. If you run the application with *Querix Phoenix*, the variable value will be set to *WTK*. *Querix Chimera* will initiate the value *JAVA* for the variable *a*.

Example

This application illustrates the work of several built-in functions as well as programmer-defined ones.

```
#####
# FUNCTION program blocks and some built-in functions
#####
DEFINE
  m_char      CHAR(80)

MAIN

DEFINE
  v1_date      DATE,
  v_integer    INTEGER,
  v1_char,
  v2_char,
  v3_char      CHAR(60),
  v_max_char   CHAR(32766),
  f_name       CHAR(5),
  l_name       CHAR(10),
  age          INT

DISPLAY "The results of programmer-defined 4GL functions execution:"
  AT 1,2 ATTRIBUTE(GREEN, BOLD)

# Function func1() is called without arguments and does not return anything
  CALL func1()
  DISPLAY "m_char      = ",m_char AT 2,2

# Function func2() is called without arguments and returns a single character
# value
  CALL func2()RETURNING v1_char
  DISPLAY "v1_char      = ",v1_char AT 3,2

# You can invoke a function which returns exactly one value without the CALL
# statement:
  LET v_max_char = 'My first phrase is '' ,func2() CLIPPED, '''
  DISPLAY "v_max_char = ",v_max_char CLIPPED AT 4,2

# Function func3(v_integer) is called with one formal argument passed to it
# and returns a single value

  LET v_integer = 37                      -- here you can specify your age
  CALL func3(v_integer) RETURNING v1_char
  DISPLAY "v1_char      = ",v1_char AT 5,2
```



```
# Function func4(v_integer) is called with three formal arguments passed to
# the function and returns three values

CALL func4(50, 1960, "Pete") RETURNING v1_char,v2_char,v3_char
                                -- arguments here are actual values and not
                                -- variables

DISPLAY "v_max_char = ",v1_char CLIPPED,v2_char CLIPPED,v3_char CLIPPED
AT 6,2

SLEEP 5
LET f_name = "Jack"
LET l_name = "Smith"
LET age = 30
CALL func5 (f_name || l_name, age)-- Two variables f_name and l_name
                                -- are combined into one by the double
                                -- pipe concatenation operator.
                                -- Then this new value is sent to the
                                -- func5 function formal argument
                                -- "fullname"

SLEEP 3

DISPLAY "The results of some Built-In 4GL Functions:"
AT 9,2 ATTRIBUTE(GREEN, BOLD)

# "Hello World" phrase is created using ASCII function
LET v1_char = ASCII 72,ASCII 101,ASCII 108,ASCII 108,ASCII 111,ASCII 32,
      ASCII 87,ASCII 111,ASCII 114,ASCII 108,ASCII 100,ASCII 33
DISPLAY "v1_char = ",v1_char AT 10,2
# "Hello World" phrase in digital format is created using ORD() function
LET v1_char = ORD("H"),ORD("e"),ORD("l"),ORD("l"),ORD("o"),ORD(" ")
      ,ORD("W"),ORD("o"),ORD("r"),ORD("l"),ORD("d"),ORD("!")
DISPLAY "v1_char = ",v1_char AT 11,2

LET v1_char = UPSHIFT("abcdefg")    -- lower case characters of the
                                -- variable are capitalized

DISPLAY "UPSHIFT (v1_char) = ",v1_char AT 12,2
LET v1_char = DOWNSHIFT("ABCDEFG") -- upper case characters of the
                                -- variable are transformed into lower
                                -- case
DISPLAY "DOWNSHIFT(v1_char) = ",v1_char AT 13,2

LET v1_char = ARG_VAL(0)           -- returns the name of the executed
                                -- program (functions.exe)
DISPLAY "ARG_VAL(0) = ",v1_char AT 14,2
LET v1_char = ARG_VAL(2)           -- returns the second command line
                                -- argument
DISPLAY "ARG_VAL(2) = ",v1_char AT 15,2
LET v_integer = NUM_ARGS()        -- returns the total number of the
                                -- arguments passed to the program
                                -- through the command line

DISPLAY "NUM_ARGS() = ",v_integer AT 16,2
```



```
SLEEP 5

CALL func6() -- this function calls some other built-in functions
SLEEP 20

END MAIN

#####
# Function func1() without arguments. It does not return values
#####

FUNCTION func1()
  LET m_char = "Hello World!" -- assigning a value to a module variable
                                -- which is visible from the function
END FUNCTION

#####
# Function func2() without arguments. It returns a single character value
#####

FUNCTION func2()
  DEFINE
    ret_value CHAR(20) -- declaring a variable for the returned value

  LET ret_value = "Hello World!"
  RETURN ret_value      -- this value will be received by the RETURNING
                        -- clause of the corresponding CALL statement
END FUNCTION

#####
# Function func3(years) with one argument. It returns a single value
#####

FUNCTION func3(years)
  DEFINE
    years      INTEGER,
    ret_value  CHAR(20) -- declaring the returned value

  LET ret_value = "I am ",years USING "<<&," years old."

  RETURN ret_value      -- this value will be received by the RETURNING
                        -- clause of the corresponding CALL statement
END FUNCTION

#####
# Function func4(years,year_birth,your_name) receives three arguments and
# returns three values
#####

FUNCTION func4(years,year_birth,your_name)
  DEFINE
    #local variables declaration
```



```
years      INTEGER,
year_birth INTEGER,
your_name   CHAR(80),

ret_string1 CHAR(60),    -- first returned value
ret_string2 CHAR(60),    -- second returned value
ret_string3 CHAR(60)     -- third returned value

LET ret_string1 = "My name is ",your_name CLIPPED, "."
LET ret_string2 = "I was born in ",year_birth USING "<<<&," year."
LET ret_string3 = "I am ",years USING "<<&," years old.

# all three values will be passed to the CALL statement, they should match
# the values in the RETURNING v1_char,v2_char,v3_char clause in order, number,
# and data types

RETURN ret_string1, ret_string2, ret_string3

END FUNCTION
#####
# Function func5(fullname, age) receives two arguments one of which is
# made of two variables by the double pipe concatenation operator
#####

FUNCTION func5 (fullname, age) -- The argument 'fullname' receives the values
-- of two variables f_name and l_name
-- concatenated by the double pipe operator
-- in the main program block

DEFINE fullname CHAR (20),
       age      INT,
       str      STRING
LET str = "      PERSONAL INFORMATION:      "
# The blank spaces added by the programmer at the beginning and the end of
# the string str are removed by means of the trim()function before displaying

DISPLAY trim(str) AT 7,2 ATTRIBUTE (green, bold)

# The CLIPPED operator only deletes automatically generated blank spaces

DISPLAY fullname CLIPPED, " is " , age, " years old" AT 7,24

END FUNCTION
#####
# Function func6() receives no arguments and demonstrates practical
# application of some built-in functions described in the chapter
#####

FUNCTION func6()

DEFINE
rand_val,
```



```
copy_rand_val,  
gui_or_char    INT,  
ui_type       CHAR (4)  
  
LET gui_or_char = fgl_fglgui() -- The fgl_fglgui() function is used  
-- to determine the mode in which the  
-- application is running  
DISPLAY "The mode is (GUI - 1, CHARACTER - 0): "   AT 17,2  
ATTRIBUTE (GREEN, BOLD)  
  
DISPLAY gui_or_char AT 17,39  
LET ui_type = fgl_getuitype() -- This function is invoked in the  
-- GUI mode in order to get information  
-- about the current type of the user  
-- interface  
DISPLAY "The current user interface is: " AT 18,2 ATTRIBUTE (GREEN, BOLD)  
DISPLAY ui_type AT 18,32  
  
LET rand_val = fgl_random(20,50) -- The program will generate a random  
-- number in the range from 20 to 50  
DISPLAY "A random number in the range from 20 to 50 is: " AT 19,2  
ATTRIBUTE (GREEN, BOLD)  
  
DISPLAY rand_val AT 19,48  
  
END FUNCTION
```



Date and Time Variables

When you create a program in 4GL, you sometimes need to refer to date and time data. 4GL supports three simple data types for storing chronological information. Two of them are used to store points in time, and one stores time spans.

DATE Variables

The DATE data type is used to store calendar dates. The value of a DATE variable is stored as an integer that indicates how many days have passed since December 31, 1899. If you need to refer to the date before DECEMBER 31, 1899 the integer value will be negative. The default display format of the DATE value depends on the features of your system. If the system uses the U.S. English locale, the default DATE display format is:

mm/dd/yyyy

where *mm* stands for a month (the value may range from 1 to 12), *dd* stands for a day (the value may range from 1 to 31), and *yyyy* stands for a year (the value may range from 0001 to 9999). You can assign a value to a DATE variable in the following way:

```
LET my_date = "11/02/1988"
```

If you enter only one or two symbols for the *year* part, the program will automatically fill the missed ones according to your program settings which depend on the DBCENTURY environment variable described later in this chapter (by default, it will put the current century and decade). If you want to store a date belonging to another century, you must specify all the four digits. If you want to specify a year that belongs to the first century AD, you should put zeros before the year identifier:

```
LET my_date = "11/02/0088"
```

You can't assign a value that indicates years BC to a variable of the DATE type.



Note: There is one more way to assign a value to a DATE variable. You can input the DATE variable value as an integer, e.g.

```
LET my_date = 868
```

This means, that value of *my_date* variable will get the value, equal to the date of the 868th day after the December 31, 1899, i.e., 05.18.1902.

DATETIME Variables

The DATETIME data type is used to store more concrete information about a moment of time than the DATE variables do. Such variables can contain not only the year, month, and day, but also the specified time of the day. The DATETIME variables are stored as DECIMAL values.

The DATETIME Qualifier

The *DATETIME qualifier* specifies which time units a DATETIME variable should include. The syntax of the DATETIME qualifier is as follows:



`first TO last[(precision)]`

where *first* and *last* stand for units of time, such as YEAR, MONTH, DAY, HOUR, MINUTE, SECOND, or FRACTION. The *precision* is applicable only to the FRACTION units which follow the TO keyword, it specifies how many digits there should be specifying the fractional part of the seconds. The scale is fractional, if you omit its specification, the fractional part will consist of three digits.

The ranges of meanings for the time units are shown in the table below.

Keyword	Time Unit	Range of Values
YEAR	A year	0001 (A.D.) – 9999 (A.D.)
MONTH	A month of the year	1 – 12
DAY	A day of the month	1 – 31
HOUR	An hour of the day	0 – 23
MINUTE	A minute of the hour	0 – 59
SECOND	A second of the minute	0 – 59
FRACTION	A decimal fraction of the second.	from one to five digits with three being the default value.

The syntax of the DATETIME variable declaration is as follows:

`DEFINE datetime_variable DATETIME first TO last[(precision)]`

In practice this will look something like the following:

```
DEFINE date_t DATETIME YEAR TO MINUTE,
        date_t2 DATETIME HOUR TO FRACTION(5)
        ...
```

As the result, the *date_t* variable will be declared as containing all the time units in the range from year to minutes (these are year, month, day, hour, and minute) and the *date_t2* variable all the time units in the hour-to-fraction scope (these are hour, minute, second, and fraction consisting of 5 digits). None of the time units within the specified range can be omitted when assigning values and their places cannot be changed; the range should remain consecutive.

There are several features you should keep in mind when specifying a DATETIME qualifier:

- If no year is specified in the DATETIME qualifier and it has the MONTH keyword at the beginning, 4GL refers to the system date to restore the additional information about the year, if needed.
- If your qualifier begins with DAY, you can get unexpected results, unless the month, specified by your system, has 31 days.
- The *first* and the *last* time units may be of the same size, but the *last* one cannot be larger than the *first* one (e.g.: SECOND to YEAR). If it is, the program will produce a compilation error.

	Note: You cannot specify the <i>DATETIME qualifier</i> after the DISPLAY statement. If you do, the program will produce a compile-time error. If you need to display a DATETIME value, you should first assign it to a variable.
---	---



DATETIME Literal

There are two ways to assign a value to a DATETIME variable. You can assign it as a *DATETIME literal* or as a *character string*.

To assign a DATETIME literal, use the DATETIME keyword followed by whole numbers and appropriate separator symbols enclosed in parentheses. The whole numbers represent date and time values. The syntax is as follows:

```
DATETIME(YYYY-MM-DD Hh:mm:ss.fff)first TO last
```

where *MM* – month, *DD* – day, *YYYY* – year, *hh* – hour, *mm* – minute, *ss* – second, *fff* – second fraction. The date (YYYY-MM-DD) and the time (Hh:mm:ss.fff) values should be separated by a space. The default precision for year, month, day, hour, minutes, and seconds is set for 2, except for the year (its precision is 4), but it can be changed.

	Note: To separate time units, you can use only the delimiters, specified in the syntax
--	---

An example of the valid DATETIME variable declaration and value assignment is given below:

```
DEFINE date_t DATETIME YEAR TO FRACTION(2)  
  
LET date_t = DATETIME (2010-06-14 12:58:47.56) YEAR TO FRACTION(2)
```

If you use DATETIME literals, you can assign all the time units declared before or only those ones which you need. For example, if you have declared a variable as YEAR to SECOND, you may assign a value as MONTH to MINUTE, YEAR to DAY, etc, but you can't miss any one of the time units between the *first* and the *last* one, and you can't assign values to separate time units, for example, to *month* and *minute* only.

Here is an example of a partial value assigned:

```
DEFINE date_t1 DATETIME YEAR TO FRACTION(2)  
  
LET date_t1 = DATETIME (06-14 12:58) MONTH TO MINUTE
```

If you want to assign a year between 1 and 99, put zeros before these digits.

If you don't assign some of the units that have been declared before, the program restores the missed information automatically. The time units that are larger than the first specified ones will get the values, taken from the system calendar. The time units that are smaller than the last specified one will be assigned to zeros. If you want the DATETIME value to include only one time unit, the *first* and the *last* time units should be identical (e.g.: DAY to DAY).

DATETIME Character Strings

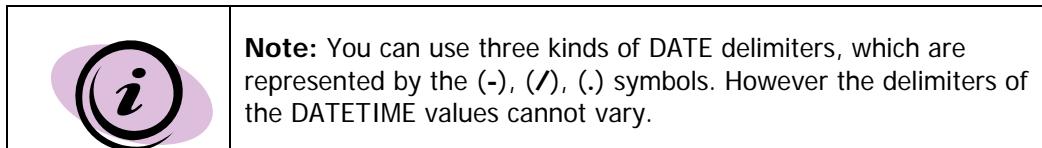
The second way to specify a DATETIME value is to use a *character string*, which is also called a *numeric datetime value*. You can use a character string to specify any set of time units, provided that you mention all the time units declared by the DATETIME qualifier in the DEFINE statement. The character string has to be enclosed in single (') or double (") quotes, but it doesn't have to be preceded with the DATETIME keyword.



If you declared a variable *my_datet* as DATETIME YEAR TO SECOND, you can assign value to it in the following ways:

```
LET my_datet = "2010-06-11 17:55:54"  
LET my_datet = '2010-06-11 17:55:54'
```

The program will produce an error, if you don't specify one or several of the declared time units, if you omit a delimiter, or if you add an extra blank space to the string.



Note: You can use three kinds of DATE delimiters, which are represented by the (-), (/), (.) symbols. However the delimiters of the DATETIME values cannot vary.

INTERVAL variables

The INTERVAL data type is used to store differences between two points in time. A time span is stored as a DECIMAL value consisting of a contiguous sequence of values that represent time units. You can use the INTERVAL data type to store any quantities that are usually represented by units of time, for example, age, sums of ages, time periods required for some activities, person-hours of effort, etc. The INTERVAL value doesn't depend on the actual date. The INTERVAL data can have zero or negative values.

The INTERVALS also have their qualifiers which are similar to the qualifiers of the DATETIME. They are also declared as

```
first[(precision)] TO last[(precision)]
```

but as you can see, they have a little different issue of precision which is discussed later.

The intervals are divided into two groups depending on the qualifiers used. These types of intervals need different qualifiers and numeric values, so the 4GL cannot convert an interval of one type into an interval of another type:

	Year-Month Intervals	Day-Time Intervals
Time spans stored	Spans of years, months, or both	Spans of days, hours, minutes, seconds, fractions of second, or contiguous subsets that include all or several of them
Qualifier keywords used	YEAR, MONTH	DAY, HOUR, MINUTE, SECOND, FRACTION
Declaration example	DEFINE yml INTERVAL YEAR TO MONTH	DEFINE yml INTERVAL DAY TO SECOND

The INTERVAL Qualifier Precision

Any time unit except FRACTION located before the TO keyword can have *(precision)*. The precision of such time units is optional and is 2 by default (excluding YEAR). It can be set in the range from 1 to 9 and it



specifies how many digits there can be in the largest time unit. It is useful, if you have to assign or calculate large time spans. E.g. if you need to assign a time span containing 215 days, 21 hours and 15 minutes to a variable of the INTERVAL data type, you need to declare this variable as follows:

```
DEFINE my_interval INTERVAL DAY(3) TO MINUTE
```

If you do not specify the precision for DAY qualifier in such case, you will be able to store only 99 days in this variable.

The YEAR has the DEFAULT precision 4. The FRACTION cannot have the precision in the position before TO, because in such case you will have FRACTION TO FRACTION qualifier where the precision is specified for the FRACTION keyword following TO. The FRACTION (precision) can acquire values from 1 to 5.

```
DEFINE  
    var1 INTERVAL MONTH(9) TO MONTH,  
    var2 INTERVAL HOUR(5) TO FRACTION(5),  
    var3 INTERVAL FRACTION TO FRACTION(2)
```

The INTERVAL Literal

There are two ways to assign a value to an INTERVAL variable. They are assigning an *INTERVAL literal* and *INTERVAL character_string*.

To assign a value to an INTERVAL in literal format, you should keep to the following syntax:

- For *year-month* INTERVAL:

```
LET variable = INTERVAL (YYYY-MM) first TO last
```

Where *YYYY* stands for the year and the *MM* stands for the month. The *first* and *last* qualifier values can be either YEAR or MONTH.

You can specify the precision only for the first part of the qualifier. For example, the interval having the qualifier YEAR(2) TO MONTH can be specified as follows:

```
LET year_month = INTERVAL (10-06) YEAR(2) TO MONTH
```

- For *day-time* INTERVAL:

```
LET variable = INTERVAL (DD hh:mm:ss.fff) first TO last
```

Where *DD* stands for the day, *hh* stands for the hour, *mm* stands for the minutes, *ss* stands for the seconds, and the *fff* stands for the fractions of seconds. The *first* and *last* qualifier values can be any of the following: DAY, HOUR, MINUTE, SECOND, or FRACTION.

You can specify an INTERVAL having the qualifier DAY(3) TO HOUR as follows:



```
LET day_hour = INTERVAL (100 12) DAY(3)TO HOUR
```

This interval spans 100.5 days.

As with DATETIME variables, it is not necessary for you to assign values for all the time units you have declared before.

INTERVAL Character String

You can also specify the INTERVAL value as a character string, which is similar to DATETIME character string. To do it, you shouldn't use the INTERVAL keyword, but you should enclose the value in single ("") or double ("") quotation marks. You omit the INTERVAL qualifier, but you must use the delimiters:

```
DEFINE int_1 INTERVAL YEAR(2) TO MONTH  
LET int_1 = "05-06"
```

The example above shows how to specify a period of time that is 5 years and 6 months long. The INTERVAL character string must describe all the time units that have been declared in the INTERVAL qualifier. If you miss any time unit, the program will produce an error.

Format of Date and Numeric Values

The default format of the DATE and DATETIME values depends on the environment variables of your system. The following variables influence these values:

- DBDATE - defines the format of the dates displayed;
- DBCENTURY - defines how the one- and two-digit year specifications within DATE and DATETIME values will be abbreviated

There are also environment variables which influences the format of the numeric values:

- DBMONEY - defines the format of the MONEY values
- DBFORMAT - defines the format of the numeric values such as DECIMAL INTEGER SMALLFLOAT FLOAT MONEY SMALLINT

We will first describe these variables and then show how their values can be changed.

Date Formats

The DBDATE variable applies only to DATE values and defines the following:

- The order of the month, day, and year time units within a date
- Whether the year is printed with two digits (Y2) or four digits (Y4)
- The time-unit separator between the month, day, and year

Its value has the following structure:

MD | DM Y4 | Y2 / | - | . | 0

or



Y4 | Y2 MD | DM / | - | . | 0

- MD stands for month-day sequence
- DM stands for day-month sequence
- Y2 stands for a two-digit year specification
- Y4 stands for a four-digit year specification
- /, -, ., or 0 represent the time-unit separator, where 0 means that no time separator will be used.

The default value for the U.S. English locale is "MDY4/". To specify that you want the date format in day-month-year format with 2 digits for the year specification and a full stop as a separator you should assign the following value to this variable: "DMY2.".

The DBCENTURY format influences not the display format directly, but the extension of a year entered only partially. It can have one of the following values which are case insensitive and depend on the system clock:

- R - the value will be prefixed with the digits of the current century (default)
- C - uses the nearest date of the past, future or current century to expand the value
- F - uses the nearest date of the future century to expand the value
- P - uses the nearest year of the past century to expand the value

Here are some examples of the DBCENTURY variable influence:

Abbreviated date	System date	R	C	F	P
1/1/50	04/06/2010	01/01/2050	01/01/2050	01/01/2050	01/01/1950
1/1/10	11/11/1999	01/01/1910	01/01/2010	01/01/2010	01/01/1910
1/1/0	11/11/1999	01/01/1900	01/01/2000	01/01/2000	01/01/1900

Numeric Formats

The DBMONEY environment variable affects only the MONEY values and is very convenient for setting the default currency symbol and its position, as well as some other features. The structure of its value is as follows:

[front] .|, [back]

The front or back is the position of the currency symbol and the symbol itself. By default it is the front dollar sign (\$). The comma or the full stop are the decimal delimiters which will separate decimal places. The default separator is a full stop. Thus the default value of this variable is "\$."

To change the default currency symbol you should specify it either before the separator or after it, depending on where you want it to occur - before or after the value. Thus, if you want to set the euro symbol as the default currency symbol and to place it after the money value, you should assign the following to the variable: ".€".

The DBFORMAT environment variable sets the general pattern for numeric values, including MONEY values. Its value has the following structure:



```
*|front : *|thousands : decimal : *|back
```

Here either front or back stands for the currency symbol and its position, just like in the DBMONEY variable and it is applicable only to the money values. Thousands stands for the default thousands separator, decimal - for the default decimal separator. Asterisks (*) can be specified instead of any fragment except decimal, if you do not want to specify the symbol for this option. Columns are just separators between the options. The default value of this variable is "*;*:.;*". It will result in the following display: 1 900 782.78.

Thus to specify that numeric values should have a comma as the decimal separator and a full stop as the thousands separator without specifying the currency symbol, you should assign the following value: "*;.;*". It will result in the following display: 1.900.782,78.

Setting Environment Variables

You can find out the current value of any environment variable by using fgl_getenv() function. It takes a character expression argument and returns a character string, both of which are the value of an environment variable.

The argument of fgl_getenv() must be a character expression that returns the name of an environment variable. To evaluate a call to fgl_getenv(), 4GL takes the following actions at runtime:

If the requested environment variable exists, the function returns it as a character string and then returns control of execution to the calling context. The identifier of an environment variable is not a 4GL expression, so you typically specify the argument as a quoted string or character variable. For example, this call evaluates the DBMONEY environment variable:

```
fgl_getenv( "DBMONEY" )
```

To set a new value to a variable, you should use fgl_putenv() function. This function allows the developer to set environment values in the 4GL processes environment. Thus you can change the values of the environment variables at runtime. This function takes a character string as its argument. It should contain the variable name, the equal sign and the new value.

Here are several examples which set the new values for the variables described above:

```
CALL fgl_putenv( "DBDATE=DMY4." )
CALL fgl_putenv( "DBCENTURY=C" )

CALL fgl_putenv( "DBMONEY=,£" )

CALL fgl_putenv( "DBFORMAT="*:,.:€" )
```

Time Operators and Functions

There are a number of operators and functions that influence values of date and time variables.

An *operator* is a command that forces program to change value or a data type of a variable or to return a certain value. A *function* causes a program perform a number of actions. The functions used with date and time values are built-in functions which function pretty much the same as operators.

In 4GL, built-in functions and operators are very similar. One and the same keyword may be regarded as a function or as an operator, depending on whether it has arguments or not, thus the only difference is whether or not it has the parentheses.



The TODAY Operator

The TODAY operator refers to the system clock and returns its value to a DATE variable. Thereby, the DATE variable gets the value, which represents the current date. The TODAY operator may be useful, if you don't need the current time value. The time of execution and the system clock accuracy are important for the correct functioning of the TODAY operator. Here is an example of the TODAY operator in use:

```
DEFINE my_date DATE  
LET my_date = TODAY
```

This part of the source code will assign the current date to the *my_date* variable.



Note: If you need to refer to the current date, it is not necessary to assign it to a variable. You can use the TODAY operator within some statements. For example, the following line of the source code will display the current date:

```
DISPLAY TODAY at 2,3
```

The CURRENT Operator

The CURRENT operator is used to return both the current day and time from the system clock. The value returned by this operator is a DATETIME value which has the default or a specified precision. The DATETIME qualifier requirements are similar to those of the [The DATETIME Qualifier](#) of the DATETIME data type. The DATETIME Qualifier is optional, and you can use it to specify the precision of the value returned by the CURRENT operator. If you don't specify any qualifier, the default precision will be set, which is YEAR TO FRACTION(3).

Here is an example of the CURRENT operator usage. The current date and time will be assigned to the *cur_dt* and displayed to the screen:

```
DEFINE cur_dt DATETIME YEAR TO MINUTE  
LET cur_dt = CURRENT  
DISPLAY cur_dt
```

If FRACTION is the last time unit, you can specify the scale within the range of 1 to 5 (e.g. LET *a* = CURRENT SECOND TO FRACTION(4))



Note: If you use the CURRENT operator more than once in a statement, it can return different values every time.

The UNITS Operator

The UNITS operator is used to convert integer expressions to INTERVAL values. If the value you want to convert to the INTERVAL value has a fractional part, the fractional digits will be ignored by 4GL. The INTERVAL value created by the UNITS operator stores only one unit of time, specified after the UNITS keyword (e.g. YEAR TO YEAR, MONTH TO MONTH, DAY TO DAY, etc.).



You can use the UNITS operator to change the values of the DATE, DATETIME and INTERVAL variables easily, without specifying additional time values. For example, if you need to find the date of a project completion, you may add the number of the days required for it to the date of the project start:

```
DEFINE start_date, finish_date DATE,  
       days_needed INTEGER  
LET start_date = "06/16/2010"  
LET days_needed = 200  
LET finish_date = start_date + days_needed UNITS DAY
```

This part of the source code will assign the value "*01.02.2011*" to the *finish_date* variable which is the result of addition of an INTERVAL containing 200 days to the *start_date* variable.

If you use the UNITS operator as a part of a left-hand arithmetic operand, you must include the operand in parenthesis:

```
LET finish_date = (days_needed UNITS DAY)+start_date
```

The EXTEND() Function

The EXTEND() function is used to change the precision and scale of DATE and DATETIME variables. The function syntax is as follows:

```
EXTEND (date_and_time, qualifier)
```

The DATETIME qualifier of the EXTEND() function has the form *first* TO *last* and its declaration doesn't differ from the qualifier declaration described above. The qualifier of the EXTEND() function is used to specify the precision and scale of the argument expression. The qualifier and the expression are separated with a comma and both must be taken in parentheses following the EXTEND keyword.

```
LET born_on="11/28/1979"  
LET my_birthday = EXTEND(born_on, MONTH TO DAY)
```

The example above will remove the year value and return "11-28"

If you don't specify the precision and scale, the default values will be set. Thus, the default qualifiers for the EXTEND() function are:

- YEAR TO FRACTION (3) for a DATETIME argument
- YEAR TO DAY for a DATE argument

The DATE and DATETIME values can have any precision allowed by 4GL, but there is a list of restrictions for the values represented by character strings:

- The character string mustn't have the DATE format (e.g. "17/06/2010")



- It mustn't include numeric DATETIME values, whose time units are impossible to recognize accurately that is without delimiters or only with delimiters which can apply to different time units (e.g. "17:02", "10", 11-08)
- The character string expression that is an argument of the EXTEND() function, must not return an INTERVAL value.

The TIME Operator

The TIME operator is used to convert the *time of day* part of the DATETIME value to a character string. The string has the format *hh:mm:ss*, based on a 24-hour clock. The *hour* (*hh*), *minute* (*mm*), and *second* (*ss*) values are stored as 2-digit strings. The TIME can be used:

- As an operator (without the parentheses and arguments) – in such case it will return the current time from the system clock.
- As a function (with an argument in parentheses) - in such case it will return the time part of the value of the argument.

Here are two examples of the TIME operator usage. In the first one, TIME operator returns the *time* value from the system clock, in the second the TIME function returns a *time* value from a variable value.

```
DEFINE
    time_v CHAR(80),
    dt DATETIME YEAR TO FRACTION(5)
LET time_v = TIME          -- returns current time
LET dt = "2010-06-14 17:12:24.12345"
LET time_v = TIME(dt)      -- returns "17:12:24.12345"
```

The time of execution and the system clock accuracy are important for the correct functioning of the TIME operator.

The DATE() Operator

The DATE operator is used to convert CHAR, VARCHAR, DATETIME, or integer value to a DATE value. The value returned by the DATE operator is stored as a character string. It functions like the TIME operator, but returns the date part instead of the time part of a value. The DATE can be used:

- As an operator (without the parentheses and arguments) – in such case it will return the current date from the system clock in the format weekday, month, day, year.

	Note: The DATE operator without arguments returns a character value. You cannot assign it to a variable of DATE or DATETIME data types.
--	--

- As a function (with an argument in parentheses) - in such case it will return the date part of the value of the argument in the format mm/dd/yyyy. This value can be assigned to DATE variables.

Here is an example of a DATE value assignment to a CHARACTER variable:

```
DEFINE
    date_ch CHAR(80),
```



```
dt DATETIME YEAR TO MINUTE,
date_v DATE
LET dt = "2010-06-14 17:12"
LET date_ch = DATE(dt) -- returns "2010-06-14"
```

The DATE operator without parentheses and arguments returning the current date, returns it in the format weekday, month, day, year:

Element	Description	Size	Example
Weekday	an abbreviation of the name of the day of the week	3 characters	Mon
Month	an abbreviation of the name of the month	3 characters	Jan
day	the number of the day of the month	2 digits	15
Year	the year	4 digits	1999

Here is an example of value returned from the system clock:

```
DISPLAY "The current date is: ", DATE at 2,3
```

On the Tuesday, June 15, 2015, this line of the source code will produce the following character string:

```
The current date is: Tue Jun 15 2015
```

The DATE operator can also convert a character, integer, or DATE value, if it represents the number of days that have passed since the December 31, 1899.

```
DEFINE date_v CHAR (20),
LET date_v = DATE(40021) -- date_v will contain "7/28/2009" value
```

The accuracy of the DATE operator functioning depends on the system clock accuracy and on the time of the operator execution.

The DAY(), MONTH(), YEAR(), WEEKDAY() Functions

The DAY(), MONTH(), and YEAR() functions are used to extract a value of the corresponding time unit from DATE or DATETIME values. The values returned by these functions are positive integers, representing the number of the day, the month, or the year, respectively. The WEEKDAY() function returns a positive integer ranging from 0 to 6 and indicating the day of the week that corresponds to the date, determined by a DATE or DATETIME value. So, 0 will represent Sunday, 1 – Monday, 2-Tuesday, etc.

The functions may be useful when you need to perform arithmetic operations to the *day, month, or year* values, because the integer values are easier to manipulate. The general view of the DAY function is:

```
DAY (argument)
```

where *argument* is a DATE or DATETIME expression. Other functions have similar syntax.

Here is an example of the functions application in a program:

```
DEFINE date_v DATETIME YEAR TO SECOND,
        day_v, month_v integer
LET date_v = DATETIME (10-06-16 15:23:11) YEAR TO SECOND
LET day_v = DAY(date_v)           -- gets value "16"
```



```
LET month_v = MONTH(date_v)      -- gets value "06"
```

You can use CURRENT and TODAY operators as arguments of the DAY(), MONTH(), YEAR(), WEEKDAY() functions.



Note: You cannot use these functions after the DISPLAY statement. If you need to display a result of a function execution, you must first assign it to a variable, and then display the variable.

The USING Operator

We already know that the USING operator can be used to format numeric values. It can also be used to change the format of the DATE values. By default, a date is displayed in the mm/dd/yyyy format in the U.S. English locale. You may need it to have different representation in different parts of your program. This does not require any environment changes. The USING operator deals with it easily. The syntax of the USING operator is:

```
date USING "pattern"
```

where *date* is a value of DATE data type or a character string representing a date and *pattern* is a set of specific symbols specifying the format. Here are the available symbols for the pattern:

Pattern Symbol	Effect	Range of Results
dd	Day of the month in 2 digit format	1 through 31
ddd	Day of the week in 3-letter abbreviation	Sun through Sat
mm	Month in 2 digit format	1 through 12
mmm	Month in 3-letter abbreviation	Jan through Dec
yy	Year in 2-digit format	trailing digits 00 through 99
yyy	Year in 4-digit format	0001 through 9999

Any other symbols are treated as literals, thus you can change the order of the elements and specify any delimiters you like. Here are several examples:

```
LET date_v = "08/26/2011"
DISPLAY date_v USING "yy mm dd" AT 1,2      -- displays "11 08 26"
DISPLAY date_v USING "dd-mm-yyyy" AT 2,2 -- displays "26-08-2011"
DISPLAY date_v USING "ddd, dd mmmm, yyyy" AT 3,2 -- displays "Fri, 26 Aug,
2011
```

Arithmetic Time Expressions

The time values can be operands of arithmetic expressions. That's how we can find intervals, evaluate how much time have passed or will pass, estimate the deadlines, etc. You can use four standard arithmetic operators: +, -, /, *. The result of the expression depends on the data types of the operands:

- DATE values are returned when you add or subtract DATE values and integer expressions



- DATETIME values are returned when you add INTERVAL value to DATE or DATETIME value; or when you subtract an INTERVAL from a DATE or DATETIME.
- INTERVAL values are returned in such cases: when using any of the four operators if both operands are INTERVALS, when subtracting DATE or DATETIME from DATE or DATETIME.

	Note: You can use / and * operators only in the expressions returning INTERVAL values.
--	---

A time or a date value used as an operand of an arithmetic expression mustn't have the form of a quoted string or a numeric date and time values or intervals. You should use only DATETIME or INTERVAL literals instead. Here is an example of the appropriate usage of time values in arithmetic operations:

```
DEFINE time_span INTERVAL YEAR TO MONTH
LET time_span = DATETIME (2010-11) YEAR TO MONTH
    - DATETIME (2008-07) YEAR TO MONTH
                                # returns "2-04" - 2 years and 4 months
```

Arithmetic time expressions have a number of peculiarities which you should remember in order to prevent mistakes and invalid results.

- Be careful when you perform arithmetic operations on Year-Month intervals and DATE values or when you add or subtract a month interval from any calendar date. The 4GL can produce wrong dates when performing arithmetic to values representing the date that is later than the 28th (e.g., it may produce such date as February 30 or June 31).
- If the precision of the estimated result of an arithmetic expression includes time units, that are not present in one or several operands, you have to use the EXTEND() function in order to adjust the format of the operands values to the format of the estimated result. For example, you cannot add an INTERVAL of 3 days and 6 and a half hours to a DATETIME value with the precision DAY TO HOUR, but you can use the EXTEND() function in order to do it:

```
DEFINE a DATETIME DAY TO SECOND
LET a = EXTEND( DATETIME (12 12) DAY TO HOUR, DAY TO SECOND)
    + INTERVAL (3 6:30:00) DAY TO SECOND
```

Date and Time Variables Conversion

As you could see, arithmetic expressions and built-in functions may cause or need conversion of data type for date and time variables. The general rules of conversion described in the previous chapter are in effect for these variables, but there is still a number of principles you should remember if you want the program to convert DATE, DATETIME, and INTERVAL values:

- The DATE values can be converted into any other data type except INTERVAL. If a DATE value is converted into a DATETIME value, an implicit EXTEND(*value*, YEAR TO DAY) function is invoked by 4GL.
- The DATETIME values may be converted into CHAR/VARCHAR/STRING, DATETIME or BIGINT values.



- The INTERVAL values can be converted into CHAR/VARCHAR/STRING values and INTERVAL values with different qualifiers.
- The INTEGER, SMALLINT, BIGINT, DECIMAL, MONEY, FLOAT, and SMALLFLOAT values can be converted into any other data type except DATETIME and INTERVAL.
- If there is any difference between DATETIME or INTERVAL qualifiers of the converted and the resulting values, 4GL may cut the resulting values from the left or right.
- If any numeric value is converted into a DATE value, the result is corresponding to the number of days that have passed since December 31, 1899.

Example

This application illustrates date and time variables with different precisions, as well as different ways of assigning values to these variables.

```
#####
# The usage of the time data types: DATE, DATETIME, and INTERVAL
#####

MAIN

DEFINE
    v1_date, v2_date DATE,
    v1_datetime      DATETIME YEAR TO MONTH,          -- this variable can include
                                                               -- only year and month values
    v2_datetime      DATETIME DAY   TO SECOND,        -- this variable can include
                                                               -- day, hour, minute, and second
                                                               -- values
    v3_datetime      DATETIME HOUR  TO HOUR,         -- this variable can include
                                                               -- only hour values
    v4_datetime      DATETIME YEAR  TO FRACTION(5),   -- this variable can include all kinds of
                                                               -- time units: year, month, day, hour,
                                                               -- minute and fraction
    v1_interval      INTERVAL YEAR TO MONTH,          -- this variable can include only
                                                               -- year and month values
    v2_interval      INTERVAL DAY   TO FRACTION(5),   -- this variable can include all kinds of
                                                               -- time units: year, month, day, hour,
                                                               -- minute and fraction
    v3_interval      INTERVAL HOUR TO SECOND,        -- this variable can include only
                                                               -- hour, minute and second values
    v_char           CHAR(80),
    v_integer         INTEGER

#####
# Part one: assigning values
#####

DISPLAY "It shows the default value for uninitialized variables:" AT 1,2
ATTRIBUTE(GREEN, BOLD)

DISPLAY "v1_date = ", v1_date AT 2,2
-- should be set TO 12/31/1899 by default
-- when not initialized
```



```
DISPLAY " v1_datetime = ",v1_datetime AT 2,25
        -- set to NULL when not initialized
DISPLAY " v1_interval = ",v1_interval AT 2,45
        -- set to NULL when not initialized
SLEEP 3

DISPLAY "Here the values are assigned to the variables of the DATE data type:"
AT 3,2 ATTRIBUTE (GREEN, BOLD)
LET v1_date = 1      -- here we assign the minimum possible day in such way,
                     -- as the date values are stored as integers: 01/01/1900
DISPLAY "v1_date DATE = ",v1_date AT 4,2

LET v2_date = "12/31/10" -- here we assign a date with incomplete year
                         -- notification, 4GL extends it to the current
                         -- century by default: 12/31/2010
DISPLAY "v2_date DATE = ",v2_date AT 4,32

SLEEP 3

DISPLAY "Here the value is assigned to the variable of the DATETIME types:"
AT 5,2 ATTRIBUTE (GREEN, BOLD)

LET v2_datetime = "16 15:30:55" -- here we assign a date DAY TO SECOND
DISPLAY "v2_datetime = ",v2_datetime AT 6,2

LET v1_datetime = DATETIME(1980-11)YEAR TO MONTH -- here we assign a
                                                -- datetime literal value
DISPLAY "v1_datetime YEAR TO MONTH = ",v1_datetime AT 6,32

SLEEP 3

DISPLAY "The values are assigned to the variables of the INTERVAL data types:"
AT 7,2 ATTRIBUTE (GREEN, BOLD)

LET v1_interval = "9999-11"      -- here we assign an interval which
                                 -- includes years and month
DISPLAY "v1_interval = ",v1_interval AT 8,2

LET v2_interval = "99 23:59:59.99999" -- here the maximum possible
                                         -- interval value is assigned
DISPLAY "v2_interval = ",v2_interval AT 8,32

LET v3_interval = INTERVAL(36:15:07)HOUR TO SECOND -- here we assign an
                                                 -- interval literal value
DISPLAY "v3_interval  INTERVAL(value)DAY TO SECOND = ",v3_interval AT 9,2

SLEEP 3
#####
# Part two: operators and functions
#####

DISPLAY "Operators: TODAY, DATE, TIME, CURRENT:" AT 10,2
ATTRIBUTE(GREEN, BOLD)
```



```
LET v1_date = TODAY          -- the today's date is assigned
DISPLAY "TODAY = ",v1_date AT 11,2

LET v_char = DATE           -- the current date is assigned in the format:
                           -- weekday month day year
DISPLAY "DATE = ",v_char AT 11,23

LET v_char = TIME            -- the current time is assigned in the format:
                           -- hh:mm:ss
DISPLAY "TIME   = ",v_char AT 12,2

LET v_char = CURRENT         -- The current date and time are assigned in the
                           -- format: YYYY-MM-DD hh:mm:ss.fff
DISPLAY "CURRENT  = ",v_char AT 12,23

SLEEP 5

DISPLAY "Functions: YEAR(), MONTH(), DAY(), WEEKDAY(), MDY():" AT 13,2
ATTRIBUTE(GREEN, BOLD)

LET v_integer = YEAR(TODAY)      -- the current year is assigned
DISPLAY "YEAR(TODAY) = ",v_integer AT 14,2

LET v_integer = MONTH(TODAY)     -- the number of the current month
                           -- (from 1 to 12) is assigned
DISPLAY "MONTH(TODAY) = ",v_integer AT 14,30

LET v_integer = DAY(CURRENT)     -- the current day (from 1 to 31) is
                           -- assigned
DISPLAY "DAY(CURRENT) = ",v_integer AT 14,60

LET v_integer = WEEKDAY(TODAY)    --the current day of week is assigned
                           -- (0 - Sunday, 1 - Monday, etc.)
DISPLAY "WEEKDAY(TODAY) = ",v_integer AT 15,2

LET v1_date = MDY(6,20,2010)      -- operands are converted into a date
                           -- value in the month-day-year order
DISPLAY "MDY(6,20,2010) = ",v1_date AT 15,30

SLEEP 5

DISPLAY "The EXTEND() operator:" AT 16,2 ATTRIBUTE(GREEN, BOLD)

LET v3_datetime = CURRENT
LET v2_datetime = EXTEND(v3_datetime, DAY TO SECOND)
                           -- the datetime value is extended using the
                           -- information from the system clock
DISPLAY "v2_datetime, DAY TO SECOND= ",v2_datetime AT 17,2

LET v1_date = TODAY
LET v1_datetime = EXTEND(v1_date, YEAR TO MONTH)
                           -- the datetime value is contracted to two time
                           -- units, days are discarded
DISPLAY "v1_date, YEAR TO MONTH= ",v1_datetime AT 17,45
```



```
SLEEP 5

DISPLAY "Here are some arithmetic operations (+, -) and UNITS operator:" AT
18,2 ATTRIBUTE(GREEN, BOLD)
LET v1_interval = 2010 UNITS YEAR      -- the interval contains now 2010 whole
                                         -- years (no additional days)
DISPLAY "v1_interval = ",v1_interval AT 19,2

LET v3_interval = 45 UNITS MINUTE      -- the interval now contains 45 whole
                                         -- minutes, 0 hours and 0 additional
                                         -- seconds
DISPLAY "v3_interval = ",v3_interval AT 19,30

LET v1_date = DATE("02/28/2003")      LET v2_date = DATE("02/29/2004")
LET v_integer = v2_date - v1_date    -- the integer stores the number of days
                                         -- in year 2004
DISPLAY "v_integer = ",v_integer AT 19,60

LET v1_date = DATE("01/01/1900")      LET v2_date = DATE("12/31/1900")
LET v1_datetime = TODAY - 1 UNITS MONTH -- we subtract one month from
                                         -- the current date
DISPLAY "v1_datetime = ",v1_datetime AT 20,2

LET v3_datetime = EXTEND(CURRENT,HOUR TO HOUR) + 2 UNITS HOUR
                                         -- we add two hours to the current hour value
DISPLAY "v4_datetime = ",v3_datetime AT 20,30

# the value of the interval is current datetime (in hour to second format) minus
# 10 hours and minus ten more minutes and contains: 9 hours, 50 minutes and 00
# seconds
LET v3_interval = EXTEND(CURRENT,HOUR TO SECOND) -
                  EXTEND(CURRENT - 10 UNITS HOUR,HOUR TO SECOND) -
                  10 UNITS MINUTE
DISPLAY "v3_interval = ",v3_interval AT 20,55

LET v1_date = DATE("01/07/1983")      -- Here you can set the date of your birth
LET v1_interval = EXTEND(TODAY,YEAR TO MONTH) - EXTEND(v1_date,YEAR TO MONTH)
                                         -- then you will see the interval which
                                         -- you have lived so far
DISPLAY "You have already lived:",v1_interval, " (in years and months)"
AT 21,2 ATTRIBUTE(YELLOW, BOLD)

SLEEP 10

DISPLAY "The USING() operator with time data:" AT 22,2 ATTRIBUTE(GREEN, BOLD)

LET v1_date = TODAY
DISPLAY "TODAY USING ""dd"" = ",v1_date USING "dd" AT 23,2
                                         -- only the day of the month as a 2-digit
                                         -- number is displayed
DISPLAY "TODAY USING ""ddd"" = ",v1_date USING "ddd" AT 23,30
                                         -- the day of the week as a 3-letter
                                         -- abbreviation (Sun through Sat) is described
DISPLAY "TODAY USING ""mmm"" = ",v1_date USING "mmm" AT 23,55
```



```
-- Month as a 3-letter abbreviation
-- (Jan through Dec) is displayed

#The USING operator can change the format of the data displayed:
DISPLAY "TODAY USING ""dd.mm.yyyy"" = ",v1_date USING "dd.mm.yyyy" AT 24,2
# You can put a comment indicator before this line and remove one from one
# of the following lines in order to see how the other formats look like
-- DISPLAY "TODAY USING ""yyyy-mm-dd"" = ",v1_date USING "yyyy-mm-dd" AT 24,2
-- DISPLAY "TODAY USING ""mm/dd/yy"" = ",v1_date USING "mm/dd/yy" AT 24,2

SLEEP 10
END MAIN
```



4GL Windows

A 4GL program is typically used to produce output to a computer monitor (there are options to produce the output to a file or to a printer, but they have limited scope of usage and are discussed much later). The output is performed to a specific part of your monitor screen. In Windows™ the output is displayed in a command line window. In UNIX/Linux the output is performed to the command line.

The command line window or the console to which an application sends the data is called the 4GL Screen. It is the basic area which can display all kinds of information sent by the program.

The Notions of the 4GL Screen and 4GL Windows

In Windows™ the parameters of the 4GL Screen are determined by the parameters of the Command Line tool. If you run the program in the character mode, the 4GL screen cannot display any graphic objects - it can display only printable characters. If you use the GUI mode, graphic objects (for example, pictures) can be displayed to the screen. You cannot use the 4GL Screen as a command line to enter the commands while a 4GL program is executed. If you have several 4GL applications launched, each of them will have its own 4GL screen.

It has the standard size, which can be modified in the same way as you modify the size of the Windows™ Command Line window: right-click on the header, select 'Properties', 'Layout' tab and set the width and height. Other settings applied to the Command line tool also influence the 4GL Screen.

4GL Windows

You can display the information directly to the 4GL Screen. However, it is considered to be more convenient to organize the information by means of the 4GL windows. A 4GL window is a rectangular area on the 4GL Screen which has a definite name, size, and position. All 4GL windows of a program must be located within the 4GL screen; they cannot exceed the limits imposed by it.

When a 4GL application starts, it has only the 4GL Screen all of which is considered to be its currently active window. Then a program can open a number of windows, to which it will display some information. These windows can overlap, thus hiding some pieces of information displayed in the lower level windows. They can be closed, brought to the top; they also can have different display attributes.

Opening a Window

There are two ways to create a window. First is creating a window with the OPEN WINDOW statement. The second way is to call the *fgl_window_open()* built-in function.

Opening a window with a statement

The OPEN WINDOW statement declares the name of the window you want to open, its position and size, however, only the window name is the required section. The syntax of this statement is as follows:

```
OPEN WINDOW window_name AT line, column  
WITH length ROWS, width COLUMNS
```



- The *window_name* is the identifier of the window, opened by the statement. It must be unique among the windows of the same program. It cannot include whitespaces.
- The *line* and *column* are the coordinates of the top left corner of the window on the 4GL Screen. They must be integers or any variables or expressions that return integer values. The AT clause of the OPEN WINDOW statement behaves very much the same like the AT clause of the DISPLAY statement. It tells the program on which row and at which column the top left corner of the window must appear.
- The *length* and *width* are also integers or any expressions that return integer values. The *length* specifies the number of the rows the window must have. The *width* specifies the number of columns. They both specify the size of the window and are to be used together.

The WITH clause specifies the dimensions of the window. Here is an example of the OPEN WINDOW statement with the WITH clause:

```
# the window "my_win1" is opened on the 10th column of the 5th line  
# of the 4GL Screen; it is 15 lines long and each line can include  
# up to 45 characters.  
  
OPEN WINDOW my_win1 AT 5,10 WITH 15 ROWS, 45 COLUMNS
```

The AT clause cannot be omitted - it specifies the position of the window.

Opening a window with a function

The *fgl_window_open()* function is synonymous to the OPEN WINDOW statement and has the same effect. The window name and parameters are passed to the function as arguments. Here is the general syntax of this function:

```
fgl_window_open(window_name, line, column, height, width, border)
```

The window name can be represented by a quoted string or a character variable. The border argument can have values "true" - which enables the border, and "false" - which disabled the border. The border specifies that the window will or will not have the outline distinguishing the window area from the background. In 4GL the border of a window is enabled using the BORDER attribute which is described below.

The following lines of the source code are equal in their effect:

```
OPEN WINDOW my_win1 AT 5,10 WITH 15 ROWS, 15 COLUMNS  
  
CALL fgl_window_open("my_win1",5,10,15,15, false)
```

4GL windows do not have any border by default, thus the window area defined by the program can be identified by a user only if something is displayed to it. To make the window border visible in the character mode, you can use the display attributes.



Window attributes

Window display attributes have the syntax common to all ATTRIBUTE clauses. In 4GL there are a number of statements which can acquire ATTRIBUTE clauses. Some of them can contain specific attributes used only with the specified statement - i.e. BORDER attribute for the OPEN WINDOW statement. The DISPLAY statement has the most common attribute clause which includes the display attributes applied to all statements which can have attributes. The succession of the attributes is not important for any attribute clause.

Thus the ATTRIBUTE clause of the OPEN WINDOW statement can contain all the attributes used for the DISPLAY statement and the BORDER attribute. They are:

- **Colours:** WHITE, YELLOW, RED, MAGENTA, BLUE, GREEN, CYAN, BLACK
- **Intensity attributes:** NORMAL, DIM, BOLD
- **Type face attributes:** UNDERLINE, REVERSE, BLINK, INVISIBLE
- **BORDER attribute**

The syntax is also the same:

```
ATTRIBUTE (attribute [, attribute])
```

However, the OPEN WINDOW statement can have some other attributes not applied to the DISPLAY statement. It can have the BORDER attribute, which creates a visible border around the window, e.g.:

```
OPEN WINDOW my_win2 AT 7,10 WITH 10 ROWS, 25 COLUMNS  
ATTRIBUTE (YELLOW, BORDER)
```

The example above will open window *my_win2* at the 10th column of the 7th row of the screen which is 10 lines long and 25 characters wide. You will be able to see this window. In the character mode it will have yellow border 1 pixel wide. If you run the program in a GUI mode, the new window will be surrounded by a graphical border and will have a window header which includes the window name and the *close*, *maximize*, and *minimize* buttons which are absent, if a window is opened without the BORDER attribute. The colour attribute doesn't influence the GUI window border.

All the information displayed to a window with a colour attribute (both in GUI and character modes) will be displayed in the specified colour, because the colour attribute affects the whole window.

	Note: The OPEN WINDOW statement also can acquire some other attributes which specify the position of some objects or messages within the window. These attributes are discussed later in this document.
--	--

If you use the *fgl_window_open()* function to create a window, you can add the border by replacing the *false* keyword with the *true* keyword. Below is the example of creating a window with a border by means of the *fgl_window_open()* function:

```
CALL fgl_window_open( "my_win1" ,5,10,15,15, true)
```



When you add a border to a window, it needs some place to be added to. The border is added outside the window. It means, that if you specify the window location as 2,2, the upper left corner of the border will have the coordinates 1,1. So, if you plan to add a border to your window, make the coordinates not less than 2, 2, otherwise, the window won't fit the screen. The border specification does not influence the actual window size if the program is run in a GUI mode.

	<p>Note: You cannot specify the colour attributes for a window opened with the help of the function.</p>
--	---

Displaying to a Window

When a window is opened, it automatically becomes the current window. All the output performed by statements (e.g. the information displayed by the DISPLAY statement) following the OPEN WINDOW statement is performed to that window until 4GL encounters any of the following:

- Another OPEN WINDOW statement opening one more window – in such case the new window will become the current one and from this moment the output will be performed to the new window.
- The current window is closed (closing windows is discussed later in this chapter) – in such case the further output will be performed to the window which has been current before the closed one, or to the screen, if no other window is opened.
- Another window is made the current window explicitly – in such case the output is done to the new current window.

When you display something to a window, you should specify the coordinates within that window, where 1,1 is the top left corner of the window. Keep in mind that the width of a window can be less than that of the screen, thus some displayed values which fit the screen may not fit the window. The right bottom corner of the window will have coordinates *length, width* (as specified in the WITH *length* ROWS, *width* COLUMNS clause of the window declaration). Thus you cannot display something to a window, if any of the coordinates are larger than the window size.

Attributes precedence

The DISPLAY attributes of a window can be overridden by the attributes of a DISPLAY statement which displays the information to this window. The next example illustrates the precedence of the attribute clauses:

```
#First we open a window with green colour attribute and a border
OPEN WINDOW my_win3 AT 2,2 WITH 15 ROWS, 60 COLUMNS
ATTRIBUTE ( BORDER, GREEN)

#The following DISPLAY statements will display the data
#to the opened window
```



```
DISPLAY "First string" AT 1,2
DISPLAY "Second string" AT 2,2 ATTRIBUTE(WHITE)
DISPLAY "Third string" AT 3,2
```

The example above will open a window with green border and display "First string" and "Third string" in green. But the "Second string" will be displayed in white, because the attributes of the DISPLAY statement are more specific and override the window attributes, which are more general in this case. In 4GL, more specific attributes (which apply to a narrower area) always override more general attributes (which apply to a wider area).

Current Window

You can open more than one window. Remember, that the names of the windows must not match. If you specify the same coordinates in their AT clauses, the second window will overlap the first one. If their size is the same, the first window will not be seen at all.

All the windows open in the 4GL Screen compose a so called window stack. Each new window is put to the top of the stack. The window on the top of this stack is considered the current window where the input and output are performed. The current window also covers any parts of the other windows or the information on the 4GL Screen, if they overlap.

You can make any window from the stack current. This means that it will be brought to the top and will be available for input/output. There are two ways to do it: the CURRENT WINDOW IS statement and the *fgl_window_current()* function. The first way is to use the following syntax:

```
CURRENT WINDOW IS window_name
```

Here the *window_name* is the identifier of a window which has been previously opened by the OPEN WINDOW statement. Below is an example showing interaction of several windows open in the same program:

```
OPEN WINDOW win1 AT 2,2 WITH 15 ROWS, 30 COLUMNS ATTRIBUTE (BORDER)
DISPLAY "win1" AT 2,2

OPEN WINDOW win2 AT 4,4 WITH 15 ROWS, 30 COLUMNS ATTRIBUTE (BORDER)
DISPLAY "win2" AT 2,2

OPEN WINDOW win3 AT 7,7 WITH 15 ROWS, 30 COLUMNS ATTRIBUTE (BORDER)
DISPLAY "win3" AT 2,2

CURRENT WINDOW IS win1
```



The second way to make a window the current one is to use the `fgl_window_current()` function. Its effect is the same as the effect of the CURRENT WINDOW statement. The syntax of the function invocation is as follows:

```
CALL fgl_window_curent ("window_name")
```

To make the window `win1` from the previous example current, you can replace the CURRENT WINDOW statement with the following line:

```
CALL fgl_window_curent ("win1")
```

Below are the schematic stack representations, they show the order of the windows in the virtual window stack. The window stack is just an order in which 4GL stores windows; it does not reflect the overlapping effects:

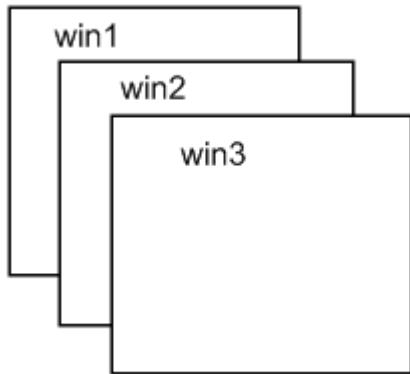


Figure 1

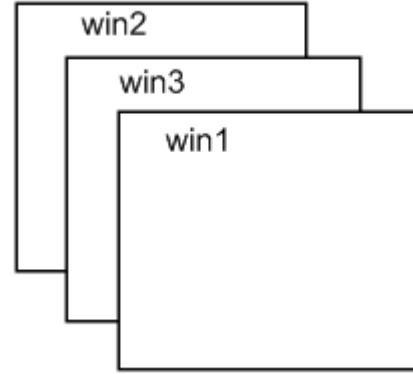
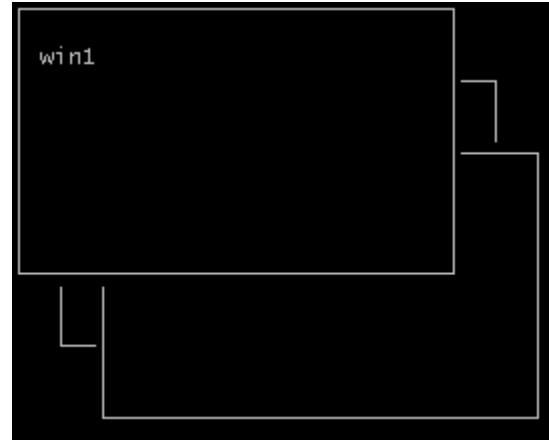
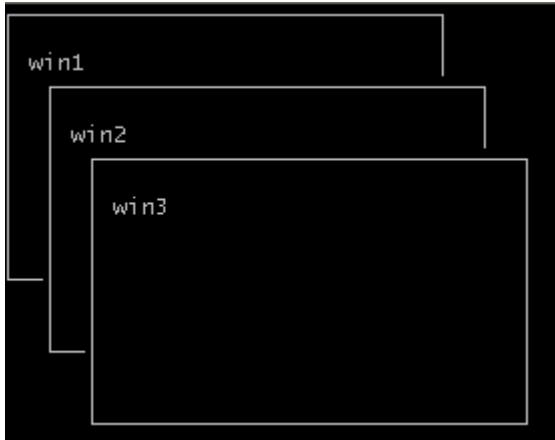
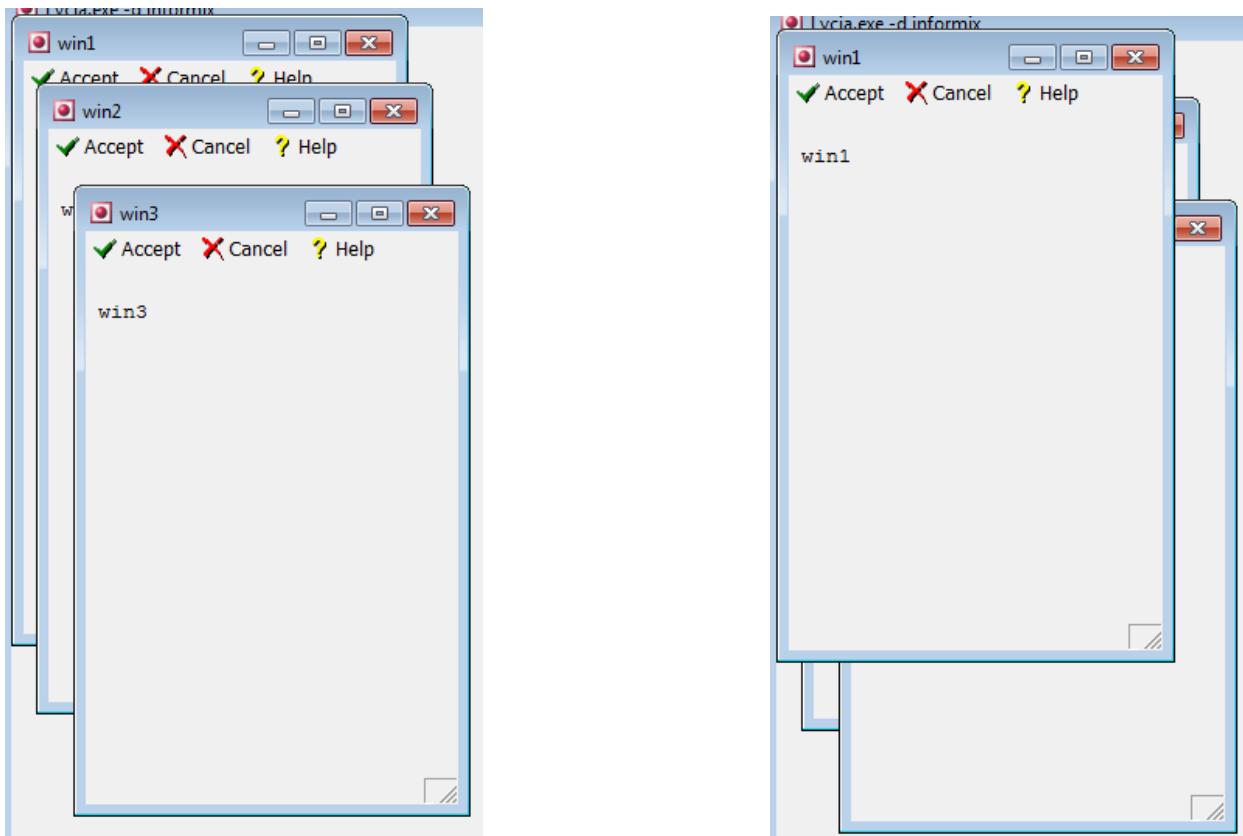


Figure 2

Here is the visual result of the stack being changed (the program is run in the character mode):



If you run the program in a GUI mode, you will see the following:



Notice, that the windows forms in character mode and in GUI mode are different even though the specified delimiters are the same. This happens because column width and row height are different in character and GUI modes.

If you run a program in a GUI mode, you can manually change the position of the window. To do it, click on the window header, hold the left mouse button, and drag the window to a new place on the screen.

You can make the 4GL or GUI Screen the current window, if you use the following code:

```
CURRENT WINDOW IS SCREEN
```

Closing and Clearing Windows

You can clear a window. Clearing a window will erase everything displayed in it. However, the variables retain their values, even if they are not displayed anymore, so they can be redisplayed later with another DISPLAY statement. As with specifying a current window, you can use a built-in function or a statement if you want to clear or to close a window. To clear a window use:

```
CLEAR WINDOW window_name
```

Or

```
CALL fgl_window_clear ("window_name")
```

Here the *window_name* is the identifier of a window which has been previously opened by the OPEN WINDOW statement or the *fgl_window_open()* function.



If you want to clear the entire screen, you need to specify the SCREEN keyword after the CLEAR statement. Clearing screen removes all the information from the screen, including the information in all the windows. If you want to clear only the screen and leave the information in the windows, use the CLEAR WINDOW SCREEN statement.

If you do not plan to use a window anymore throughout your program, you can close it. When the window is closed, it is deleted from the window stack and all information displayed in it disappears (but the values of the variables remain unchanged). To close a window:

```
CLOSE WINDOW window_name
```

Or

```
CALL fgl_window_close ("window_name")
```

Here the *window_name* is the identifier of a window which has been previously opened by the OPEN WINDOW statement or by the *fgl_window_open()* function.

If you want to clear or close a window, you don't have to make it the current one.

A 4GL Window as an Object

Querix4GL offers another way of working with 4GL windows. To create and manipulate a window, you can define a variable of the WINDOW data type.

The syntax for declaring a WINDOW data type variable doesn't differ from a standard declaration statement for variables of the simple data types:

```
DEFINE variable_name WINDOW
```

For example, this line of code declares a WINDOW data type variable with the name *window_1*:

```
DEFINE window_1 WINDOW
```

The declaration above actually creates a WINDOW object which describes a window. This object can be passed freely to and from functions. A WINDOW variable can be declared with any scope of reference like other 4GL variables.

Manipulating window objects

To manipulate an object that describes a 4GL window, a number of methods can be called. However, due to the object-oriented nature of this data type, the syntax for linking a WINDOW variable with these methods differs from that used in built-in 4GL functions. Nevertheless, these methods can be called like normal 4GL functions either using the CALL statement or using the method linked with a variable as an operand of a 4GL expression.

A method is linked to a WINDOW object by means of the dot (.) operator following the name of a variable, and only then specifies a particular method with appropriate arguments, if required. If there are no arguments needed, the empty parentheses should still be present:



CALL *variable_name*.*method*([*arguments*])

Here *variable_name* stands for the name of a WINDOW data type variable, *method* – for the name of a method called on that variable, and *parameters* – for any parameters that the method takes. The *variable_name* and *method* are bound together by the dot operator.



Note: The case in a method name is not important. It can be either upper or lower case.

Methods Used with a WINDOW Variable

As has already been said, to manipulate a WINDOW object several Querix4GL methods can be called. They are described in the following sections.

Opening a Window

To open a new 4GL window of a specified size at a specified position the *Open()* method is used. The general syntax for opening a window by means of this method is as follows:

```
CALL variable_name.method_name( "window_name",
                                row_number,
                                column_number,
                                num_of_rows,
                                num_of_columns [, 
                                "window_attributes" ] )
```

The *Open()* method takes six parameters separated by commas:

- *window_name* - is either a quoted string which defines the name of the window, or a character variable which receives this name. The window name doesn't have to coincide with the name of the variable you declared in the DEFINE statement. In addition, you may decide to leave the new window without a name. In this case, however, you still **must** include the quotation marks. Otherwise, a compile-time error will be produced.
- *row_number* and *column_number* - are integers which represent the row and column position at which the window will open.
- *num_of_rows* and *num_of_columns* - are also integers, they define the size of the window – the number of rows and columns correspondingly.
- *window_attributes* - is a quoted string or a character variable containing such string which includes window display attributes like border or color. This parameter is optional. Also, you should remember that if the BORDER attribute is left out, the window frame will not be visible. If the arguments string contains misprints, they will not be spotted at compile time and will not produce an error at runtime. However, the argument with the misprint will not be applied, whereas the attributes with the correct syntax will still take effect.

Here is a typical call to the *Open()* method:



```
# First, we define a variable of the WINDOW data type
    DEFINE window_1 WINDOW
# The Open() method is called on window_1.
    CALL window_1.Open("", -- Although we don't find it necessary to give
                      -- the new window a name, we still have to keep
                      -- the quotation marks in place
                      2,-- The line number at which the window will open
                      2,-- The column number at which the window will open
                      20, -- The window will be 20 rows high
                      30, -- and 30 columns wide
                      "BORDER, GREEN")-- The attributes of the window
```

Displaying Data to a Window

To display some information at a particular position within a window described by a WINDOW object, you can call the *DisplayAt()* method. The syntax of the statement used to invoke this method is this:

```
CALL variable_name.DisplayAt ("displayed_data", line_number,
                               column_number, "attributes")
```

As can be seen, the method takes four comma-separated parameters:

- *displayed_data* - can be either a quoted character string you want to be displayed, or a variable you define and initialize earlier in your program. This parameter can remain empty, though you must keep the quotation marks.
- *line_number* - represented by an integer, indicates the line number where the data will appear.
- *column_number* - an integer as well, stands for the initial column position of *displayed_data*.
- *attributes* – includes standard display attributes such as border, color, etc. The attributes must be enclosed in quotes and separated by commas. It is an optional parameter. If the arguments string contains misprints, they will not be spotted at compile time and will not produce an error at runtime. However, the argument with the misprint will not be applied, whereas the attributes with the correct syntax will still take effect.

Here are some examples of the *DisplayAt()* method in use. In this code fragment the information to be displayed is represented by a quoted character string with the starting position row 2, column 2. The string will also be displayed in green and bold.

```
DEFINE window_1 WINDOW
CALL window_1.Open("My window", 2, 2, 20, 30, "BORDER")
CALL window_1.DisplayAt("The message", 2, 2, "GREEN, BOLD")
```

And here we first define a variable that will store the data to be displayed, initialize it, and then use it as the first parameter of the *DisplayAt()* method.

```
...
DEFINE a INT
LET a = 5
...
CALL window_1.DisplayAt (a, 2, 2)
..
```



Closing a Window

To close a window opened with the *Open()* method, method *Close()* is called. It takes no arguments. It frees the WINDOW variable and you will be able to associate it with another window. Until the *Close()* method is called, you cannot open another window using the same WINDOW variable.

```
DEFINE window_1 WINDOW
CALL window_1.Open("", 2, 2, 20, 60, "")
..
CALL window_1.Close()
```

Retrieving Window Attributes

There are several functions which return the attributes of an open window such as its size or its position. These functions can only be applied to the current window and accept no parameters.

Getting Window Size

The *fgl_getwin_height()* and *fgl_getwin_width()* functions return, respectively, the height in rows and width in columns of the current window.

The following snippet of the source code opens a window and displays its parameters. In this example these functions are invoked without the CALL statement. To use them in a CALL statement, the CALL statement must contain the RETURNING clause with the variable to store the value returned by the function. Thus it is considered more convenient to use such functions as operands and not as a part of a CALL statement.

```
MAIN

DEFINE width INTEGER

OPEN WINDOW w_test AT 4, 4 WITH 20 ROWS, 50 COLUMNS

ATTRIBUTE (BORDER)

LET width = fgl_getwin_width() -- returns 50

DISPLAY "Width: ", width , " columns" AT 5, 5

DISPLAY "Height: ", fgl_getwin_height(), " rows" AT 7, 5 -- returns 20

SLEEP 3

END MAIN
```



There is also a function that returns both window height and window width. It is the *fgl_winsize()* function which returns two integer values indicating the number of rows and columns that a specified window contains. The argument of the function is the name of the window which is to be evaluated, but it is optional. If the function has no argument, it will return the size of the current window. This function can be used in two ways:

- With the CALL statement. This method requires the RETURNING clause containing exactly two variables for the returned values. The first variable will obtain the number of rows and the second will contain the number of columns.

```
CALL fgl_winsize(window_name) RETURNING var1, var2
```

- As a right-hand argument of the LET statement. This LET statement must assign the values to a program record consisting of two members, these values will be the height and the width of the window respectively, e.g.:

```
LET recl.* = fgl_winsize(window_name)
```

In the following example, the program evaluates the parameters of the window and opens a form in it, if there is enough space for this form. If the window is too small to accommodate a form, the program will open a new window with the form:

```
DEFINE rws,clns INT
OPEN WINDOW w1 AT 2,2 WITH 10 ROWS, 20 COLUMNS

ATTRIBUTES (BORDER)

...
...
...

CALL fgl_winsize("w1") RETURNING rws, clns

IF rws<20 AND clns<40 THEN

    OPEN WINDOW w2 AT 3,3 WITH FORM "my_form"

ATTRIBUTES (BORDER)

SLEEP 2

ELSE
```



```
OPEN FORM my_f FROM "my_form"
```

```
DISPLAY FORM my_f
```

```
SLEEP 2
```

```
END IF
```

```
...
```

```
...
```

Getting Window Position

The *fgl_getwin_x()* and *fgl_getwin_y()* functions are used to return, respectively, the column and the row at which the top left corner of the current window is situated. They can be used in the same way as the functions described above. The following example demonstrates how the window coordinates can be displayed to the screen:

```
MAIN

DEFINE w_col, w_row INTEGER

OPEN WINDOW w_test at 5,10

WITH 20 ROWS, 50 COLUMNS

ATTRIBUTE (BORDER)

LET w_col = fgl_getwin_x() -- returns 5

CALL fgl_getwin_y() RETURNING w_row -- returns 10

DISPLAY "Column: ", w_col AT 5, 5

DISPLAY "Row: ", w_row AT 7, 5

SLEEP 3
```



```
END MAIN
```

Changing Window Parameters

There are functions which allow you to change some of the parameters of the 4GL screen or a 4GL window, namely its name and size.

Setting Window Size

The *fgl_setsize()* function is used to change the default size of a 4GL window. You must specify two integer arguments when calling the *fgl_setsize()* function. These arguments specify the new number of *lines* and *columns* that a window contains respectively.

The syntax of the function invocation is:

```
CALL fgl_setsize( [ "window_name" ], lines, columns )
```

If the window name is omitted, this function resizes the 4GL screen. If the window name is present, the function will be applied to the specified window regardless of whether it is the current window. It may be useful when the default size is not enough for the displayed information to fit it, or in case there is not much information to display and you want your application look neat and to make the screen smaller.

You can use it with the window name as the argument to adjust a window to the information displayed, if one and the same window is used for different purposes.

The following snippet of a source code demonstrates the application of this function:

```
MAIN

DISPLAY "This is a default screen size" at 2,2

SLEEP 2

CLEAR SCREEN

CALL fgl_setsize (50, 10)

DISPLAY "The size was changed to (50, 10)" at 2, 2

SLEEP 2

END MAIN
```



The function has an effect only if the application is run in the GUI mode.

Setting Window Title

This function will be applied to the 4GL screen, if it is the current window, or to the other current window, because this function does not accept the window name as its argument. If the current window has no BORDER attribute, the function will affect the 4GL the nearest window in the window stack which has the border.

When you run a program, the program window gets the name of the executable file of this program. For example, if the program name is *My_prog*, the window header will look as follows:



The program name can be followed with the executable file extension. This is optional and depends on the interface you use.

However, you can change the 4GL Screen title. To do this, use the *fgl_settitle()* function. The syntax of the function invocation is very simple:

```
CALL fgl_settitle("New_window_title")
```

It is advisable that the function is used at the beginning of your source code text, just after the declaration clause. This won't let the user see how the window name is changing during the program execution.

On the other hand, you can change the window title throughout the program execution when the program needs different kinds of user interaction. In the following example, the window title will be changed twice:

```
MAIN

DISPLAY "The title is default" at 2,2

SLEEP 5

CALL fgl_settitle("New window title")

DISPLAY "The window title has been changed" at 2, 2

SLEEP 5

CALL fgl_settitle("The second title")
```

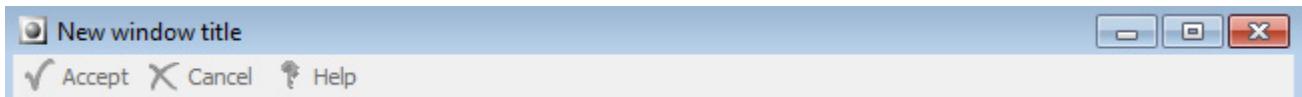


```
DISPLAY "The window title has been changed once more" at 2, 2

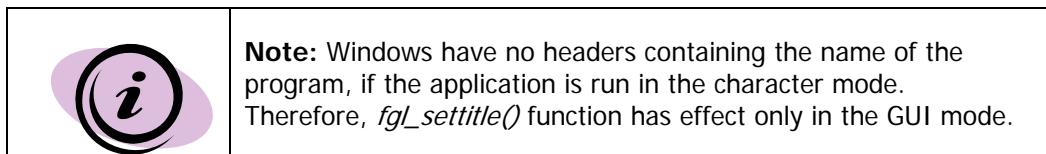
SLEEP 5

END MAIN
```

When the *fgl_settitle()* function is executed for the first time, the window title will look as follows:



When the function is executed for the second time, the window will have another title:



Suspending Program Execution

In order to make the process of working with several windows (opening, closing, clearing, etc.) more visual, we need to introduce the SLEEP statement here. The SLEEP statement should be followed by an integer which specifies the number of seconds by which the program execution will be suspended. So, to make a pause between opening two windows you can place the SLEEP statement between the OPEN WINDOW statements. In this case they will be opened one by one after the specified period of time and not all together. This usage of the SLEEP statement is illustrated in the example below. More profound description of the SLEEP statement can be found further in the manual.

Example

This application illustrates the usage of the both types of windows: explicitly and implicitly declared. It also uses the special methods to work with the former and built-in functions to work with the latter.

```
#####
# These are the examples of window management in 4GL
#####

MAIN
DEFINE x, y, wid, hgt INT, -- These variables will store values representing
                           -- the coordinates and size of the window
      win                  WINDOW -- This is a variable of the WINDOW data type

# This statement displays the string to the 4GL Screen as no window is yet
# open:
```



```
DISPLAY "These are the examples of window management: " AT 1,2
ATTRIBUTE(GREEN, BOLD)

# We open three windows consecutively, each newly opened window automatically
# becomes the current window

OPEN WINDOW w_win1 AT 6,4      -- w_win1 becomes the current window as
WITH 8 ROWS,30 COLUMNS          -- soon as it is opened
ATTRIBUTE (BORDER)              -- we need the BORDER attribute otherwise
                                -- the window border is invisible

DISPLAY "OPEN WINDOW w_win1" AT 1,1
SLEEP 1

OPEN WINDOW w_win2 AT 10,26 WITH 8 ROWS,30 COLUMNS ATTRIBUTE (BORDER)
                                -- this window overlaps w_win1, it becomes the
                                -- current window and it is put on the top of
                                -- the window stack which is now: w_win2,
                                -- w_win1
DISPLAY "OPEN WINDOW w_win2" AT 1,1
SLEEP 1
CALL fgl_window_open ("w_win3",6,48,8,30,TRUE) -- This is another way of
                                                -- opening a window.
                                                -- This window overlaps
                                                -- the two previous ones and
                                                -- becomes the current window;
                                                -- it moves to the top of the
                                                -- window stack.
                                                -- The window stack is now:
                                                -- w_win3, w_win2, w_win1

DISPLAY "OPEN WINDOW w_win3" AT 1,1
SLEEP 1

# Here we declare a current window not by opening.
# A previously opened window is made current and is moved to the top of the
# window stack.
CURRENT WINDOW IS w_win2      -- now window stack is: w_win2, w_win3, w_win1
DISPLAY "CURRENT WINDOW IS w_win2" AT 1,1
SLEEP 1
CALL fgl_window_current ("w_win1") -- We make the w_win1 window current
                                    -- by means of the gl_window_current()
                                    -- function. Now the window stack is:
                                    -- w_win1, w_win2, w_win3

DISPLAY "CURRENT WINDOW IS w_win1" AT 1,1
SLEEP 1

# Here we clear all our windows regardless of which one is the current window
CLEAR WINDOW w_win1 SLEEP 1
CLEAR WINDOW w_win2 SLEEP 1
CALL fgl_window_clear ("w_win3") -- This window is cleared using the
                                -- fgl_window_clear() function

# Now we close the windows after which only the 4GL Screen is left
CLOSE WINDOW w_win1 SLEEP 1
```



```
CLOSE WINDOW w_win2    SLEEP 1
CALL fgl_window_close ("w_win3") SLEEP 1 -- The third window is closed by
                                         -- calling the fgl_window_close()
                                         -- function

# We open new windows with different display attributes
OPEN WINDOW w_win1 AT 6,4 WITH 8 ROWS,30 COLUMNS
ATTRIBUTE (BORDER, YELLOW, BOLD)
DISPLAY "OPEN WINDOW w_win1 in YELLOW colour, BOLD" AT 1,1

OPEN WINDOW w_win2 AT 10,26 WITH 8 ROWS,30 COLUMNS
ATTRIBUTE (BORDER, REVERSE)
DISPLAY "OPEN WINDOW w_win2 in reversed video" AT 1,1

OPEN WINDOW w_win3 AT 6,48 WITH 8 ROWS,30 COLUMNS ATTRIBUTE (BORDER, MAGENTA)
DISPLAY "OPEN WINDOW w_win3 in MAGENTA colour" AT 1,1
SLEEP 2
CALL fgl_window_close("w_win1")
CLOSE WINDOW w_win2
CALL fgl_window_close("w_win3")

# And at last we clear the 4GL screen, close all the open windows and close
# the window stack
CLEAR SCREEN
SLEEP 1

CALL fgl_window_open ("w_win4", 5,5,15,60,TRUE) -- We open a new window,
CALL fgl_settitle("This is window title")           -- and set a title for it
                                                 -- with the fgl_settitle()
                                                 -- function

DISPLAY "The title of the window differs from the window name" AT 1,2 ATTRIBUTE
(GREEN, BOLD)
SLEEP 2

# Then we find the coordinates of the window using the fgl_getwin_x() and
# fgl_getwin_y() functions
LET x = fgl_getwin_x()
LET y = fgl_getwin_y()

DISPLAY "THE LOCATION OF THE WINDOW IS" AT 3,2
DISPLAY "ROW " , y, " COLUMN " , x AT 4,2
SLEEP 2

# Next we determine the size of the window. First by applying the
# fgl_getwin_x() and fgl_getwin_y() functions:
LET wid = fgl_getwin_width()
LET hgt = fgl_getwin_height()
DISPLAY "THE SIZE OF THE WINDOW IS " AT 5,2
DISPLAY hgt, " LINES BY " , wid, " COLUMNS" AT 6,2
SLEEP 2

#CALL fgl_window_clear("w_win4")
#SLEEP 1
CALL fgl_setsize("w_win4",15,40)      -- Finally, we change the size of the
```



```
-- window with the fgl_setsize()
-- function

# we can also use fgl_winsize() to get the window size
CALL fgl_winsize() RETURNING hgt,wid
DISPLAY "The window has been resized" AT 7,2 ATTRIBUTE (GREEN, BOLD)
SLEEP 2
DISPLAY "THE NEW SIZE OF THE WINDOW IS " AT 8,2
DISPLAY hgt, " LINES BY ", wid, " COLUMNS" AT 9,2
SLEEP 2
CALL fgl_window_clear("w_win4")
CALL fgl_window_close("w_win4")
SLEEP 1

# And finally, we create a new window by means of the WINDOW variable win
# defined at the beginning of the program
CALL win.Open("win1",2, 2, 15, 70,"BORDER,GREEN") -- We open the window by
-- calling the Open()method
# And display a string to it by using DisplayAt()
CALL win.DisplayAt("This window is opened with the help of a WINDOW variable",
2, 8, "CYAN")
SLEEP 5
CALL win.Close()-- The window is closed

CLEAR SCREEN

CALL fgl_setsize(15,40)
DISPLAY "The 4GL Screen has been resized" AT 2,2 ATTRIBUTE (GREEN, BOLD)
SLEEP 2
CALL fgl_winsize()RETURNING hgt,wid -- this function will return the size
-- of the 4GL Screen, as it is the
-- current window
DISPLAY "THE NEW SIZE OF THE 4GL SCREEN IS " AT 3,2
DISPLAY hgt, " LINES BY ", wid, " COLUMNS" AT 4,2
SLEEP 2
CLEAR SCREEN

END MAIN
```



4GL Form Files

Now you can display some information to the 4GL Screen or to a 4GL window, but there are some other ways to produce output and one of them is using a screen form.

A screen form is an area on the 4GL Screen or 4GL window containing some objects (form fields, character strings, etc.) in the strictly specified order. Its advantage as compared to displaying strings to a window or to the screen is that you can display a set of items repeatedly without having to format the displayed values again. Screen forms are also used for receiving input from a user, but this function will be discussed a bit later in this manual.

To use a screen form you need:

- A form specification file with .per extension where you specify what items you want to display, where to display them, and their attributes.
- A .4gl source file which is a part of a 4GL application, where you tell the 4GL to open the compiled form file and to display the form. The .4gl source code file is then used also for any form manipulations.

A Form Specification File

A form specification file can be created in the same way as a .4gl file with the help of any text editor. It must have .per extension. Any form file has obligatory sections and optional ones. A simplest form file has the following structure:

```
<DATABASE section>
<SCREEN section>
```

The DATABASE Section

The DATABASE section is used to declare against which database the form file will be compiled and which database it will use. It includes only the DATABASE keyword and the name of the database with or without server name. The snippet of code below tells the 4GL to connect to the database called "my_db" when compiling the form:

```
DATABASE my_db
```

This section is required even if you do not plan to connect to a database and your application is independent of any databases. In such case you specify the "formonly" keyword after the DATABASE keyword. It tells the 4GL that this form is to be compiled and used independently of any database, e.g.:

```
DATABASE formonly
```

As we have not yet discussed database issues, we will use the second variant of the DATABASE section: DATABASE formonly.



The SCREEN Section

The SCREEN section of a form file specifies which objects this file has and how they are located relative to one another. Here is the simplest form file which will compile, but which will not produce any visible output, because it does not contain any objects:

```
DATABASE formonly
SCREEN {
}
END
```

All form file sections can end with the END keyword, but it is optional and can be omitted. If the END keyword is omitted, 4GL considers that the previous section has ended when it encounters the keyword of the next section, or the end of the form file.

You should place the object identifiers only between the braces ({}). In the example above the form will contain no objects. The length of the form is 24 lines by default. The width of the form is defined by the position of the right-hand side of the rightmost object on the form.

You can include special keywords after the SCREEN keyword, if you want to set the maximum number of lines and the number of characters in a line of your screen form explicitly:

```
SCREEN SIZE lines BY characters
```

Specify *lines* as the total height of the form. Four lines are reserved for the system, so by default, no more than (lines - 4) lines of the form can display data. E.g.:

```
SCREEN SIZE 15 BY 40 {  
}
```

The SCREEN section can contain two types of objects:

- Static objects – cannot be changed throughout the program workflow and retain the values, shape and position throughout the program execution. They can be modified only if you edit the form specification file. These are lines, boxes and character strings.
- Dynamic objects – the values displayed to them can be changed throughout the program with the help of the code in a .4gl file. They can also accept user input. The position, shape and attributes of such objects cannot be changed without editing the form file. These objects are called form fields.

In this chapter we will discuss the creation of static form objects.

Comments in a Form File

A form file imposes some restrictions on the usage of the comment symbols due to its specific structure:

- You cannot use the sharp (#) symbol to identify comments within a form file.
- You cannot place comments into the SCREEN section of the form file.



Adding Static Objects to a Form

You can add objects only to the area between the braces within the SCREEN statement. There are mainly two types of static objects which can be added to a form, these are literal character strings (also called the static labels) and graphical objects such as rectangles or lines.

To place a character string on the form, you need to type it within the screen delimiters. Use only ASCII characters to be on the safe side. The example below will create a form with a character string displayed on it:

```
DATABASE formonly
SCREEN{
    This is my form
}
```

This string cannot be changed throughout the program execution.

You can draw a line or a box using special symbols:

Symbol	Purpose
-	Used to indicate horizontal line segments
	Used to indicate vertical line segments
p	Used to indicate the left top corner of a rectangle
q	Used to indicate the right top corner of a rectangle
b	Used to indicate the left bottom corner of a rectangle
d	Used to indicate the right bottom corner of a rectangle

For the 4GL to recognize these symbols as graphical indicators, you should insert \g symbols before the first of them in the line. To leave graphical mode, insert the same symbols (\g) after these symbols. However, the 4GL exits from graphical mode automatically at the end of a line regardless of whether you've inserted these symbols or not. This means that every line that requires graphics mode must start with the \g symbols, but does not need to end with it.

Thus to draw a line you need to type a sequence of hyphens limited by the graphical mode symbols, e.g.:

```
\g-----\g
```

To draw a box you will need all six graphical symbols. A box will look as follows:

```
\gp-----q\g
\g|           |\g
\g|           |\g
\g|           |\g
\gb-----d\g
```

The screen section with a character string, a line and a box may look like the following:

```
SCREEN{
    This is my form
    Line
    \g-----\g

    Rectangle
```

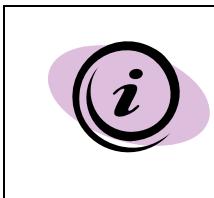
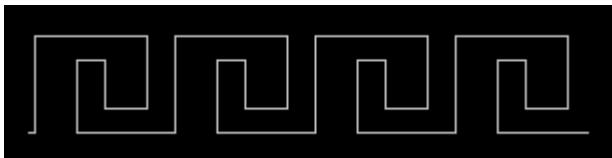


```
\gp-----q\g
\g|          |\g
\g|          |\g
\g|          |\g
\gb-----d\g
}
END
```

You can create more sophisticated patterns using these simple tools. Below is a part of the screen section which creates a meander pattern on a form:

```
\gp-----q p-----q p-----q p-----q
\g|  p-q |  p-q |  p-q |  p-q |
\g|  |  |  |  |  |  |  |
\g|  b--d |  b--d |  b--d |  b--d |
\gd b-----d b-----d b-----d b-----
```

Here is what it looks like when the application is run:



Note: To use the graphical characters as if they were normal characters, you need to assign other characters or character sequences instead of them. This is done in the system termcap or terminfo file. For that you should consult the manual that comes with your terminal. For more information refer to the Informix 4GL reference.

Restrictions

There are some restrictions as to the usage of certain characters in a form file:

- The backslash (\) cannot be included into a character string within the screen section. The 4GL attempts to interpret it as the beginning of an escape sequence, thus it is not printed.
- Your form might not compile correctly, if you attempt to use either braces ({ }) or the field delimiter symbols ([], and |) other than for their direct purposes
- The pound sign (#) or double-hyphens (--) in the screen layout as literals, not as comment indicators.

Form Fields

A form is typically used to receive the information from a user or to display it, thus the static objects described in this chapter do not play the main role in the form file. Form fields and other widgets which are the dynamic form elements play the most crucial part in a form file. The fields will be discussed in details later in this chapter, here we will give only the general overview.



A form field must be added to the SCREEN section as a string which contains the field tag and which is delimited by the brackets:

```
SCREEN{  
    [field_tag]  
}
```

Then you need to add another section to your form file where you would assign the field name to the field tag:

```
ATTRIBUTES  
field_tag=formonly.field_tag;
```

The "formonly.field_tag" is the full name of the field not linked to any database. This name is used to reference the field in the 4GL code. This name is also referenced in the script files which are discussed in the next chapter. For now you should just know that a form field is a named object on a form unlike the static labels, which are just character strings without the name a program can refer to.

Opening and Displaying a Form

Now as you know how to create a form specification file, you need to display the form in your 4GL application. The following steps should be made in order to display a form:

1. Create a form file
2. Compile it. The file extension will change from .per to .frm.
3. Create a source code .4gl file with the 4GL code which opens and displays the form (described later in this chapter).
4. Compile the source code file and link the form file to it.
5. Run the program.



Note: See Lycia Developer's Guide for the information about how to compile, link and run 4GL applications.

Let's discuss the third step in details. To tell a 4GL application which form you want to open and when you want to display it, you need to use the OPEN FORM and DISPLAY FORM statements, or built-in functions which have the same effect as the statements, but can have different arguments.

The OPEN FORM statement declares the name of a compiled form and associates this name with a form file. It has the following syntax:

```
OPEN FORM form_name FROM "form_file"
```

The *form_name* need not match the name of the form specification file, but it must be unique among other form names in the same program. Its scope of reference is the entire program. It must not contain whitespaces. The *form_file* is the name of a form specification file without the file extension, but with the relative path, if required.



The built-in function synonymous to the OPEN WINDOW statement is the *fgl_form_open()* function. The function must have two arguments when called. The first argument specifies the name of the form; the second argument specifies the form file used to open the form. Both arguments can be represented by a quoted string or by the name of character data type variable containing such string:

```
LET f_name = "my_form"  
CALL fgl_form_open(f_name, "my_form_file")
```

The function above opens a form from the file named *my_form_file* and sets the form name to *my_form*.

After you have opened a form, you can use its name to display it with the help of the DISPLAY FORM statement. It has the following syntax:

```
DISPLAY FORM form_name
```

Here the *form_name* is the name of the form previously declared by an OPEN FORM statement. Below is an example which opens and displays a form:

```
OPEN FORM form_1 FROM "source/forms/form1"  
DISPLAY FORM form_1
```

In such way the form is displayed to the currently active window (or to the 4GL screen, if it is the current window).

You can also use the *fgl_form_display()* function to display the form that has already been opened. This function requires only one argument which specifies the name of the form to be displayed. The argument can be represented by a quoted string or by a variable which contains such a string:

```
CALL fgl_form_display("my_form")
```

	Note: If a form is opened in a window, the size of the window should be large enough to accommodate the form.
--	--

Form Display Attributes

The DISPLAY FORM statement may have the ATTRIBUTE clause which determines the display attributes to be applied to the form. They override the display attributes specified for the window in which the form is opened. The ATTRIBUTE clause should follow the form name, it can contain any display attributes which we have already discussed in regard to the DISPLAY statement. An example of a form opened with display attributes is below. Though the window has its attributes, the form data will be still displayed in yellow and bold and not in green:

```
OPEN WINDOW win_for_form AT 2,2 WITH 20 ROWS, 45 COLUMNS  
    ATTRIBUTE (GREEN)  
OPEN FORM form_1 FROM "source/forms/form1"  
DISPLAY FORM form_1 ATTRIBUTE (YELLOW, BOLD)
```



Opening a Form in a Window

There is a way to open a new window together with a form. For this you can use the OPEN WINDOW statement with the WITH FORM clause. It opens a window together with a form. In this case you do not need to use the OPEN FORM and DISPLAY FORM statements.

On the one hand, it reduces the source code amount and ensures that the window matches the size of the form. On the other hand, such window can display only one form (with which it has been opened) and it cannot be larger than the form displayed, so you will not be able to display some additional information in it.

The OPEN WINDOW ... WITH FORM statement has the following syntax:

```
OPEN WINDOW window_name AT line, column WITH FORM "form_file"
```

Here the OPEN WINDOW keywords, window name, and the AT clause stick to the same rules as in other cases of the OPEN WINDOW statement described in the previous chapter. Such windows can also acquire the typical attribute clause. The WITH FORM clause replaces the WITH lines ROWS, characters COLUMNS clause of a window opened without a form. The WITH FORM keywords should be followed by the form file name which sticks to the same rules as in the OPEN FORM statement.

Position of the Form

The position of the form in a window opened in either way depends on the position of the so called form line. By default the form line is the third line of the window (or the 4GL Screen). This setting can be modified in two ways:

- You can add a special attribute to the OPEN WINDOW statement called FORM LINE.
- You can specify the position of the form by adding the OPTIONS statement to your code before the form is opened.

FORM LINE Window Attribute

The OPEN WINDOW statement can have a specific attribute FORM LINE which is included into the ATTRIBUTE clause together with other attributes. The FORM LINE attribute must be followed by an integer, or an expression returning an integer which specifies the line on which the form will start.

It works from both: windows opened with form and forms displayed to such window using the DISPLAY FORM statement. Here is an example of this attribute in use, it will cause the form to be opened on the fifth line of the window, leaving the first four free:

```
OPEN WINDOW my_win1 AT 2,2 WITH FORM "form_02"  
ATTRIBUTE (BORDER, FORM LINE 5)
```

OPTIONS FORM LINE Statement

The OPTIONS statement is used to specify a lot of display features such as lines on which the output should be performed, etc. It can specify the form line different from the default one. This statement will have effect for all the forms and windows opened after it has been executed. You can override these settings by specifying another form line for a particular window as described above. You can also use another OPTIONS statement with the same option which will override the settings of the previous one.



The example below specifies that the forms *form1* and *form3* will be opened on the second line of the windows *w1* and *w3* respectively, whereas *form2* will be opened on the fourth line of window *w2*. The same rules of attribute precedence which allow more specific attributes to override more general ones are applied here.

```
OPTIONS FORM LINE 2
OPEN WINDOW w1 AT 2,2 WITH FORM "form1"
OPEN WINDOW w2 AT 7,2 WITH FORM "form2" ATTRIBUTE (FORM LINE 4)
OPEN WINDOW w1 AT 12,2 WITH FORM "form3"
```

Graphical Elements in a Source File

We have already discussed how a line or a box can be drawn within a form-file by means of special symbols. Q4GL supports one more tool for drawing lines and boxes in a screen form. These are special built-in functions.

The *fgl_drawline()* function can be used to display a horizontal line of a specified size. It is a built-in function, but unlike the built-in functions we have already discussed in [Chapter 5](#), it needs to be called in the same way as a programmer defined function.

The syntax of the function invocation is:

```
CALL fgl_frawline (column, row, width [, colour])
```

The *column* and *row* arguments should be represented by integer values which specify the start point of the line on the screen.

The *width* argument is an integer value specifying the number of screen columns that the line is to take.

The *colour* argument is optional, it should be represented by an integer value ranging from 0 to 7 and is used to specify the colour of the displayed line. Each number corresponds to a colour in the following way:

- 0 - WHITE
- 1 - BLACK
- 2 - BLUE
- 3 - RED
- 4 - YELLOW
- 5 - GREEN
- 6 - MAGENTA
- 7 - CYAN

The function can be used to draw a line in a screen, a window, or in a window with form. If you want the line to be present in a screen or in a window, just call the function within the program code:

```
DISPLAY "0" AT 4,2
CALL fgl_drawline(4, 4, 30, 0)

DISPLAY "2" AT 6,2
CALL fgl_drawline(4, 6, 30, 2)

DISPLAY "3" AT 8,2
```



```
CALL fgl_drawline(4, 8, 30, 3)

DISPLAY "4" AT 10,2
CALL fgl_drawline(4, 10, 30, 4)

DISPLAY "5" AT 12,2
CALL fgl_drawline(4, 12, 30, 5)

DISPLAY "6" AT 14,2
CALL fgl_drawline(4, 14, 30, 6)

DISPLAY "7" AT 16,2
CALL fgl_drawline(4, 16, 30, 7)
```

This snippet of a source code will display seven coloured lines with the corresponding colour numbers to the screen.

	Note: The lines created by the <i>fgl_drawline()</i> function will be displayed in different colours only if you run your application in the character mode. The GUI mode will display all the lines in black.
---	---

You can draw a line in a form if you call the *fgl_drawline()* function just after the OPEN WINDOW ... WITH FORM statement:

```
OPEN WINDOW win_1 AT 2,2 WITH FORM "my_form"

CALL fgl_drawline(4, 4, 30, 0)
```

You can use the function, for example, to separate the form heading from the form other form contents. You can also draw a line on a form displayed by the DISPLAY FORM statement.

You can also use a special function to display a rectangle to a form or to a screen or window. The *fgl_drawbox()* function draws a rectangle of the specified size when called from the source code. The rectangle drawn by it is a part of the displayed form. Each time the corresponding DISPLAY FORM or OPEN WINDOW [...WITH FORM] statement is executed, you should also call this function, if you want to redraw the rectangle.

This function accepts more than one argument and has the following syntax:

```
fgl_drawbox(height, width, line, character [, colour])
```

Here *height* and *width* are the number of lines and characters enclosed into the drawn rectangle represented by integers - its size. The *line* and *character* are the position of the left top corner of the rectangle in the window represented by integers. The optional attribute *colour* should also be an integer specifying the colour of the displayed rectangle. The *colour* attribute can have the following values: 0 - white (default), 1 - black, 2 - blue, 3 - red, 4 - yellow, 5 - green, 6 - magenta, 7 - cyan.

Below, there is an example of a rectangle drawn by this function. This rectangle has 20 lines and 40 columns, it starts at the second character of the second line of the window and has the outline of yellow colour:



```
OPEN WINDOW win_1 AT 3,3 WITH FORM "my_form"  
  
CALL fgl_drawbox(20, 40, 2, 2, 4)
```

The colour attribute is not applied to the rectangle drawn by the function if the application is run in the GUI mode.

Closing a Form

After you finish using the form you may want to close it in order to free the memory allocated to it. Once you close the form you cannot use it in the DISPLAY FROM statement. To use it again, you should open it once more with the OPEN FORM statement. There are two ways to close a form. The first is to use the CLOSE FORM statement followed by the name of the form previously opened by the OPEN FORM statement:

```
CLOSE FORM form_1
```

The second way to close the form is to call a built-in function *fgl_form_close()*. The syntax of the function invocation is as follows:

```
CALL fgl_form_close(form_name)
```

The *form_name* can be represented by a quoted string specifying the name of the form to be closed or by a variable which contains such name. The possibility to pass a value of the variable to the function makes the function more flexible than the CLOSE FORM statement is. Below are given two ways to invoke the *fgl_form_close()* function, and both ways will lead to the same result - the program will close the form named *my_form*:

```
LET a = "my_form"  
CALL fgl_form_close(a)  
  
CALL fgl_form_close("my_form")
```

If you have displayed your form to a window and then closed this window, it doesn't mean that the form was also closed. You still have to use the CLOSE FORM statement or the *fgl_window_close()* function to close the form that has been opened before. If you don't do it, you'll be able to display the same form to another window or to the screen (the information previously displayed to this form will be erased).

The exception is when you use the OPEN WINDOW WITH FORM statement. If the form is opened this way, it can be closed by the CLOSE WINDOW statement or *fgl_window_close()* function together with the window it is opened in.

The FORM Data Type

Another way of creating and manipulating forms in Querix4GL is to define a variable of the FORM data type. The syntax for declaring a FORM data type variable doesn't differ from a standard declaration statement:

```
DEFINE variable_name FORM
```



This line creates a FORM object which describes a form. Then you can call some methods provided by Querix4GL to work with that form: open it, display some information to it, close it, etc. However, similarly to a WINDOW object described in the chapter on 4GL windows, in order to call a method on a FORM object you need to link the variable name with the method by the dot (.) operator:

```
CALL variable_name.Method([arguments])
```

The *variable_name* here is the name of a FORM data type variable you declared in the DEFINE statement, *Method* is the name of the method called on a FORM object, and *arguments* denote any parameters that the method may require.

Manipulating Form Objects

There are several built-in methods that are used to work with a FORM object. The following sections explain when they are called and what they do.

Opening a Form

To open a form described by a FORM object, the *Open()* method is called.

```
CALL variable_name.Open("form_file_name" [, "form_alias"])
```

The *Open()* method takes two parameters separated by a comma:

- *form_file_name* - represents the name of the file which contains the form you want to open excluding the extension. It can be a quoted string or a character variable.
- *form_alias* - a quoted string or a variable to which represents the name of the form. It is optional and may be used in scripting which is explained later.

Here is an example of the *Open()* method being called:

```
DEFINE window_1 WINDOW
LET f_name = "form_1" -- The variable to store the form file name
CALL window_1.Open(f_name, "f1" )
```

If an alias is not needed, you can omit it:

```
CALL window_1.Open(f_name)
```



Displaying a Form

After the form has been opened, you should display it. Otherwise, it will remain invisible. To display a form you can use the *Display()* method. The syntax of the statement required to invoke the method is:

```
CALL variable_name.Display(["display_attributes"])
```

The *Display()* method takes only one optional parameter – *display_attributes*. Here, within quotes, you can specify standard display attributes like color. These attributes override those specified when you open a new window where the form is to be displayed.

This is an example of a form displayed by means of the *Display()* method with some display attributes specified.

```
DEFINE form_1 FORM -- We define a variable of the FORM data type
..
CALL form_1.Open("formfile") -- The form is opened
CALL form_1.Display ("GREEN, BOLD") -- The form is displayed
-- The text in the form will
-- be green and bold
```

Opening a Window With a Form

As explained in the chapter on 4GL Windows, one of the ways to open a window in Querix4GL is by calling the *Open()* method on a WINDOW object. You can open and display a form to this window afterwards. However, there is a more efficient way to do that. You can open a window together with a form at a specified location without having to perform the two operations one after another. For this purpose, the *OpenWithForm()* method is called together with a WINDOW variable.

```
CALL window_vname.OpenWithForm("window_name", "form_file", line_number,
                                column_number, "window_attributes")
```

This method takes five parameters.

The first one – *window_vname* – is the identifier of the variable of the WINDOW data type you are going to work with. If you omit this parameter, you will get a compile-time error.

The second one – *window_name* – is the name of the window you are going to open, which can be represented by a quoted character string, a variable you define earlier in your program, or can be left empty. In the last case, even though there is no value, you still **must** keep the quotes in place if you want to avoid a compilation error.

The next parameter – *"form_file"* - stands for the name of the form file with the form that you want to be opened and displayed in the window. Like the *window_name* parameter, *form_file* can be a quoted character string or a CHAR variable to which you assign the form file name without the .per extension. However, in contrast to *window_name* you cannot leave the *form_file* parameter blank even by keeping the quotation marks. The program will build successfully, but a run-time error will be generated.



The next two parameters – *line_number* and *column_number* – are integers which indicate the line and column position at which the window will open.

And finally, “*window_attributes*” is a list of the standard window attributes such as border or color plus some additional ones. One of these additional attributes is the *form-line*:

form-line:<number>

It is used to point to the line within the current window at which the form will be displayed. The *<number>* value is represented by an integer. For example - “form-line: 4” will open the form at the fourth line within the current window

The *window_attributes* parameter is optional, but if you don’t include the BORDER attribute, you will not be able to see the window bounds.

Another thing worth mentioning is that with the *OpenWithForm()* method the size of the window equals the size of the form, in contrast to the *Open()* method where the window and form dimensions don’t match . Here is an example of a call to the *OpenWithForm()* method.

```
# We need to define a variable of the WINDOW data type first
...
DEFINE window_1 WINDOW
LET f_name = "form_file" -- Then we declare a CHAR variable
-- to store the form file name. Notice
-- that the extension is not included
CALL window_1.OpenWithForm ("",-- The window name is not specified,
-- but the quotation marks are kept all
-- the same
f_name,-- The variable which stores the
-- form file name
3, -- The line number where the window
-- will open
3, -- The column number where the window
-- will open
"BORDER, form-line: 4") -- The form will be
-- opened at the fourth line of the
-- current window and will have a
-- border
...
...
```

Getting the Form Size

To get the size of the form you have opened, two methods can be called: *getWidth()* and *getHeight()*. Neither of them requires parameters, and all you need to do is bind the FORM data type variable with one of these methods.



As the name suggests, *getWidth()* returns an integer which stands for the width of the form in columns, whereas an integer returned by *getHeight()* represents the height of the form in rows. You can assign these values to INT variables directly, without calling the methods explicitly.

```
...
DEFINE form_1 FORM,
wid, hgt INT
...
# The values returned by the methods are assigned to INT variables
# directly, without the CALL statement
LET hgt = form_1.getHeight() -- The hgt variable will store
-- the size of the form in
-- rows
LET wid = form_1.getWidth() -- The wid variable will store a number
-- representing the size of the form in
-- columns
DISPLAY "The form is ", hgt, " rows high, and ", wid, " columns wide"
AT 5, 5
...
...
```

Closing a Form

A form opened by the *Open()* method of the FORM object is closed by the *Close()* method. It takes no parameters, and the only thing you have to do is link the FORM data type variable with the method by the dot operator.

```
CALL variable_name.Close()
```

For example:

```
CALL form_1.Close()
```

Example

This application consists of a number of source files. For it to work normally, these source files should be associated with one program and have the specified names.

It illustrates the work with form files with both using statements and using the FORM data type with methods.

```
#####
# This example illustrates the basics of the form management in 4GL
#####

MAIN
DEFINE f form

DISPLAY "Here are the examples of screen form management: " AT 1,2
ATTRIBUTE(GREEN, BOLD)
```



```
CALL fgl_drawline(2,2,77,5) -- A horizontal line is drawn under the string
-- in the previous statement

OPTIONS FORM LINE 6                                -- all the forms displayed below till
                                                       -- the next OPTIONS statement will
                                                       -- begin at the 6th line of the 4GL
                                                       -- Screen or a window

DISPLAY "The form is opened in the 4GL Screen" AT 5,2 ATTRIBUTE(GREEN, BOLD)
CALL f.open("form_1")
CALL f.display()
sleep 2

OPEN WINDOW w_win2 AT 4,24 WITH 20 ROWS,45 COLUMNS ATTRIBUTE (BORDER)
OPTIONS FORM LINE 4                                -- the second OPTIONS statement
                                                       -- overrides the settings of the
                                                       -- first one; now the forms below will
                                                       -- be opened at the fourth line

DISPLAY "The form is opened in a 4GL window" AT 1,2 ATTRIBUTE(GREEN, BOLD)
DISPLAY "The window is bigger than the form requires" AT 2,2
ATTRIBUTE(GREEN, BOLD)

CALL fgl_form_open ("form2","form_2")    -- here we open and display the second
                                         -- form within the current window on
                                         -- its first line by means of the
                                         -- corresponding built-in functions
                                         -- instead of the standard 4GL
                                         -- statements
CALL fgl_form_display ("form2")

SLEEP 2

# After that we open a form together with a window
# This is the easiest way to open a form, if you do not plan to open another
# form in the same window
OPEN WINDOW w_win3 AT 6,47 WITH FORM "form_3" ATTRIBUTE
(BORDER,FORM LINE FIRST)                         -- we specify the form line in the ATTRIBUTE
                                               -- clause it overrides the latest OPTIONS
                                               -- statement for this window

DISPLAY " The form is opened in a 4GL" AT 4,2 ATTRIBUTE(GREEN, BOLD)
DISPLAY " window" AT 5,2 ATTRIBUTE(GREEN, BOLD)
SLEEP 2

CLOSE WINDOW w_win3                            -- we close window w_win3 opened WITH FORM and
                                              -- the form form_3 opened in it is also
                                              -- automatically closed

SLEEP 2
```



```
CURRENT WINDOW IS w_win2

DISPLAY "Now the window will be closed" " AT 1,2
ATTRIBUTE(GREEN, BOLD)
DISPLAY "But the form will remain open" " AT 2,2
ATTRIBUTE(GREEN, BOLD)

SLEEP 2

CLOSE WINDOW w_win2
-- We close the window in which a form is
-- displayed using the DISPLAY FORM statement,
-- but the form "form2" is not closed
-- automatically.
-- It remains in the memory and can be displayed
-- to another window or to the screen

SLEEP 2

OPEN WINDOW w_win2_next AT 4,24 WITH 20 ROWS,40 COLUMNS ATTRIBUTE (BORDER)
--we open another window

DISPLAY "Form2 is opened for the second time." AT 1,1 ATTRIBUTE(GREEN, BOLD)

CALL fgl_form_display("form2")-- and display "form2" into it without having
-- to open it again. Note that the same
-- built-in function we invoked
-- the first time is called again.
-- If you opened a form with fgl_form_open(),
-- you must be consistent
-- and use only built-in functions to display
-- and close the form (fgl_form_display() and
-- (fgl_form_close()) without mixing them with
-- the standard 4GL statements(DISPLAY FORM and
-- CLOSE FORM correspondingly).
-- Otherwise, some unexpected results may
-- occur.

SLEEP 2
CALL fgl_form_close ("form2")-- Now we close "form2" and release the
-- memory allocated to it.
-- It can no longer be displayed by the
-- DISPLAY FORM statement;
-- to display it again you need to use
-- OPEN FORM statement or the fgl_form_open()
-- function once more

CLOSE WINDOW w_win2_next

SLEEP 2
CALL f.close() -- we close the form opened in the 4GL Screen, it
-- cannot be displayed one more time, however, as it
-- has been displayed before, it remains displayed
-- but cannot be manipulated any more

DISPLAY "This form is now closed, but not yet cleared from the 4GL Screen"
AT 5,2 ATTRIBUTE(GREEN, BOLD)
SLEEP 2
```



```
CLEAR SCREEN -- clears the 4GL screen and removes "form1" from it
SLEEP 1

DISPLAY "Here are the examples FGL_DRAWBOX() function usage: " AT 1,2
ATTRIBUTE(GREEN, BOLD)
CALL fgl_drawline(2,2,77,5)

# FGL_DRAWBOX() displays a rectangle the top left corner of which is located
# at the 1st character of the 6th line of the 4GL Screen
DISPLAY "A rectangle is displayed to the 4GL Screen" AT 5,2
ATTRIBUTE(GREEN, BOLD)
CALL FGL_DRAWBOX(12,33,6,1,0)
DISPLAY "FGL_DRAWBOX()" AT 7,3
SLEEP 2

# FGL_DRAWBOX() displays a rectangle within form form_4, opened in window
# w_win4
OPEN WINDOW w_win4 AT 10,15 WITH FORM "form_4"
ATTRIBUTE (BORDER, FORM LINE FIRST+1)
DISPLAY "form_4" AT 1,2
CALL FGL_DRAWBOX(5,27,2,1,0)
DISPLAY "A rectangle in a form" AT 3,3 ATTRIBUTE(GREEN, BOLD)
SLEEP 2

# FGL_DRAWBOX() displays a rectangle within form form_5, opened in window
# w_win5
OPEN WINDOW w_win5 AT 6,47 WITH FORM "form_5"
ATTRIBUTE (BORDER, FORM LINE FIRST)
DISPLAY "A rectangle in a form" AT 3,3 ATTRIBUTE(GREEN, BOLD)
CALL FGL_DRAWBOX(7,35,5,4,5)
SLEEP 4

CLOSE WINDOW w_win5
SLEEP 1
CLOSE WINDOW w_win4
SLEEP 1

CLEAR SCREEN

# We can place a number of rectangles in any order, e.g. to draw letter 'V'
DISPLAY "Here are more examples of FGL_DRAWBOX() function: " AT 1,2
ATTRIBUTE(GREEN, BOLD)
CALL fgl_drawline(2,2,77,5)

CALL FGL_DRAWBOX(3,6,3,2,0)
CALL FGL_DRAWBOX(3,6,5,8,2)
CALL FGL_DRAWBOX(3,6,7,14,3)
CALL FGL_DRAWBOX(3,6,9,20,4)
CALL FGL_DRAWBOX(3,6,7,26,5)
CALL FGL_DRAWBOX(3,6,5,32,6)
CALL FGL_DRAWBOX(3,6,3,38,7)
SLEEP 5
CLEAR SCREEN
```



END MAIN

Form Files

Below is the first form file called "form_1.per". Remember that you should not specify the form file extension when referring to it in the source code. If you copy the code below to a file which you intend to use as a form, you should use the same name for the file or edit the source code where it refers to his file.

```
DATABASE formonly
SCREEN
{
\gp-----q\g
\g|\g form1\g          |\g
\g|           [ f ] |\g
\gb-----d\g
}
END

ATTRIBUTES
f = formonly.field;
END

INSTRUCTIONS DELIMITERS "  "
END
```

The second form file is called "form_2.per":

```
DATABASE formonly
SCREEN
{
\gp-----q\g
\g|\g form2\g          |\g
\g|           [ f ] |\g
\gb-----d\g
}
```



```
ATTRIBUTES
f = formonly.field;
```

```
INSTRUCTIONS DELIMITERS " "
END
```

The third form file is called "form_3.per":

```
DATABASE formonly
SCREEN
{
\gp-----q\g
\g|\g form_3\g           |\g
\g|                                [ f ] |\g
\gb-----d\g
}
```

```
ATTRIBUTES
f = formonly.field;
```

```
INSTRUCTIONS DELIMITERS " "
END
```

The fourth form file is called "form_4.per":

```
DATABASE formonly
SCREEN SIZE 25 BY 80
{
```

```
[ f ]
}
END
```

```
ATTRIBUTES
f = formonly.field;
```

```
INSTRUCTIONS DELIMITERS " "
END
```

The fifth form file is called "form_5.per":



```
DATABASE formonly
SCREEN
{
\gp-----q\g
\g|\g form_5\g           |\g
\g|               |\g
\g|               |\g
\g|               |\g
\g|\gFGL_DRAWBOX()    overlaps   form\g|\g
\g|       \gelements\g  |\g
\g|               |\g
\g|               |\g
\g|               [ f ] |\g
\gb-----d\g
}
END

ATTRIBUTES
f = formonly.field;
END

INSTRUCTIONS DELIMITERS " "
END
```



Script Options

Script options can be used, when a program is run in GUI mode. If a program is run in the character mode, they are ignored. They influence the way different objects are displayed: their position, size, colour and some other features. They are not included into the source code.



Note: Script options are applicable only if you run Phoenix or Chimera. LyciaDesktop uses the graphical Theme Designer to modify the look and feel of programs, so any script files will be ignored when run by this client.

For more information about the designer see LyciaDesktop Developers Guide and the edition of the Theme Designer Guide.

A Script File

The script options are specified in a separate file, called script file (which is a text file with **.qxs** extension), which must be located within the project and added to the program requirements. Script files can either be stored on the local machine, or the application server. In Lycia Studio the script files are created using the **File -> New -> GUI Script File** option.

As this file is not a source file, it is not processed by the compiler. Like other media files it is used by a thin client at runtime, thus any errors in this file cannot be spotted at compile-time. If a script file contains any errors, they will not cause the program to terminate, the options with errors (e.g. invalid syntax) will be ignored, and those without errors will be still executed. Script options can be placed in any order in the file, the order does not influence their execution.

When compiling a program, Lycia creates a default master script file which contains the information about the files which the program uses. The name of the master script file matches the name of the program. Therefore, the name of the programmer-defined script file should not match the name of the application; otherwise it will be overwritten with the master script file and the script options specified in this programmer-defined file won't be applied to the program.

The abilities of script files include:

- Setting defaults
- Changing colours
- Changing fonts
- Text & object manipulation
- Background images
- Toolbars & Menus

Script files should be used wherever possible for general graphical template work, i.e., for a common look and feel of colours, fonts, etc. They should also be used for configuration options and any customisation that cannot be achieved with 4GL code and Form options. Script files must be used, if the source code cannot be modified to achieve the same effect.



The Structure of a Script File

The Querix scripting language considers the 4GL display to consist of a number of components. The scripting language then allows these individual components to be configured one by one, or on a global basis.

Each program object which appearance you want to change has its own "address", which consists of:

- The name of the application it is used in,
- The name of a window of the "screen" keyword, if the object is located in the 4GL Screen
- The name of the form it is located in

Then there goes the object name. The object name generally consists of the table name and field name (i.e. rec1.f002). You must not omit the record name even if the field is not linked to any table - in this case use the "formonly" record. This is applicable only to dynamic form widgets. Static widgets and the information displayed by other means are called string objects and are discussed later in this chapter.

To apply an option to an object, you should specify its address in the order shown above; the names should be joined together by a period (.). Here is a general pattern for an address together with the object name:

```
application_name.window_name.form_name.table_name.field_name
```

Here the table name represents either a table name, if a field is linked to a database table, or the formonly keyword. The "field_name" is the name declared for the field tag. Generally, the table_name.field_name part of the address is taken from the field declaration in the [ATTRIBUTES](#) section of the form file which has been discussed shortly in the previous chapter. You cannot use a field name, if you omit the table name.

Here is an example of an address of a form field "customer.cust_id" located in from called "custom" opened in window "Screen", which is the name of the 4GL screen - the first "default" window which is opened implicitly by 4GL when a program is launched, in an application called "cms":

```
cms.Screen.custom.customer.cust_id
```

	Note: The Screen keyword is used, when the object is located on the 4GL screen rather than in a 4GL window.
---	--

You need not go all the way down to a specific object. If the address specifies only the window, like in the example below, the script option following it will be applied to the specified window. However, depending of the object type at the end of the address there may be different script options applicable.

```
cms.win2
```

A script file can also include comments. You should begin each comment line with the hash symbol (#). The comments are ignored by the thin clients. Lines consisting of white spaces only are also considered to be comments and are ignored as well.



Properties of an Object

Each object within the form has a number of properties (or script options). Not only form objects but also windows, forms themselves and other objects have their own properties which can be modified. Some of them are object unique. These resources represent aspects of the object that you may wish to modify: its position on the screen, background colour, or the control used to represent it. The name of the property must follow the object name and its address and it must be separated by a full stop (.) from them. In its turn the value of the item property must be separated from the property name by a colon:

```
address.object_name.property_name: value
```

For example deltaX and deltaY properties of objects specify horizontal and vertical offset. To specify the offset you do not need to know the actual position of an object. The resource should follow the object which you want to move. The resource accepts values, which should be separated from the resource name by a colon. In the case of deltaX and deltaY properties, the value must be an integer, specifying the number of columns and lines respectively by which the object should be offset.

The script options below move the field by 5 columns to the right and by 5 lines down from its original position in the form file:

```
cms.Screen.custom.customer.cust_id.deltaX: 5  
cms.Screen.custom.customer.cust_id.deltaY: 5
```

Here "cms.Screen.custom.customer" is the address, "cust_id" is the object name, "deltaX" and "deltaY" are the properties and "5" is the value. At the same the actual position of the field in the form file id not changed, because script options affect only the display.

Wildcard Symbols in an Address

Wildcarding features allow the objects to be more generally specified. Using wildcards you can apply a script option to a form with the same name regardless of a window it is opened in, or to fields with the same name regardless of the form and window they are used in. An asterisk (*) matches any number of arbitrary specifiers, and a question mark (?) will match exactly one specifier. However, the asterisk should be avoided where possible and the question mark should be used instead. This is because the question mark is more accurate and is more likely to generate the correct results.

The example below applies to all the objects called cust_id (window, form, field, etc.) in application cms regardless of the window, form or other objects it might belong to:

```
cms*cust_id
```

The next example applies to all fields called customer.cust_id opened in the Screen regardless of the form they belong to:

```
cms.Screen.?.customer.cust_id
```



The String Object

You may need to name not only the form objects but also any other objects in your application. The most common case is when you need to change the properties of a character string which is either a static label, or is displayed using the DISPLAY statement. Originally such objects do not have a name to which a script option can refer. However, you can search for a particular string and name it. Then script options can refer to this string using the specified name.

The syntax of the name specification is the following:

```
application.window.form.strings.string_name: character_string
```

To name a string object you should use the “**strings**” property followed by the name you want to assign. After the column you should specify the character string to which you want to assign the name. The character string must not be included into quotation marks. Here is an example:

```
cms.win1.form_22.strings.my_string: Type Control-W*
```

All the character strings displayed to the form "form_22" opened in window "win1" which begin with "Type Control-W" will be named "my_string". The asterisk here specifies that these strings can have any ending as long as they begin with the specified characters. Pay attention to the fact that this option names all the strings in the specified location which include the value specified after the column, thus a script option referring to the name given will apply to all these strings.

Because the string object name consists of one identifier and the form object name consists of two identifiers (table name and field name), the usage of the asterisk (*) in the script files should be restricted, for the GUI is not able to differentiate the object correctly.

Components with No Form

Some objects do not belong to a form, but still you need to specify them in a script file. Those objects will normally be the strings, because the fields can appear only within a form file. In such case you can use the “no-form” keyword instead of the form name in the object address. For example:

```
myapp.Screen.no-form.my_string
```

Hiding a String

The naming of a string does not apply any properties to it. One of the most common properties used with a string object is the “hidden” property. This option can be used to check whether the strings were named properly. It accepts Boolean values: “true” and “false”. When this option has value “true”, the specified string is not displayed. To apply any properties to a named string, you should use its name in a script option, e.g.:

```
app.Screen.no-form.strings.my_string: Press F1*
app.Screen.no-form.my_string.hidden: true
app.Screen.no-form.my_string.hidden: false
```



Replacing a Text String

You can replace one text string with another one by using the *text* option .The generalized structure of the string-replacing line of the script file looks as follows:

```
application_name.window name.form_name.string_name.text: text
```

The *text* stands for the character string that will replace the specified string. For the purpose of the convenience, you can use the back slash (\) symbol to terminate the current line and to make the program recognize the text on the following line as the continuation of the text. The backslash symbol won't be displayed and two lines will be displayed to the screen as one.

For example, the string named *my_str* can be replaced with another string in the following way:

```
my_appl.screen.strings.my_str: Press Control-C*
my_appl.screen.my_string.text: Click the Close\
Button to terminate the program execution and close the window
```

This will make any string starting with "Press Control-C" be replaced with the string "Click the Close button to terminate the program execution and close the window".

The Default Object

One special object which the clients recognise is called default. It differs from other objects, while it does not need the address specified. It should be specified at the beginning of a script string.

The default resource configures a number of global application settings. Most of these settings configure the appearance and behaviour of the running application. Some of the options are only effective if the thin-clients are invoked with the script, (that is, if you opened the script file from a file manager or browser). For example the options below will only take effect, if the application is invoked with the script.

```
default.host: my_server
default.port: 1689
```

The "host" and "port" are the properties of the default object. Different properties can accept different types of values. For more information about running the application on the GUI server see the corresponding thin clients' manuals and the "Graphical Clients Reference".

Here are some properties which can be applied to the default object:

Properties	Description	Type of value
default.host	The server for the application to connect to	Character string

Example: default.host:my_server

default.port	The port for the application to connect to	Character string
---------------------	--	------------------



Example: default.port: 1689

default.command	The command to execute on the application server with or without arguments	Character string
------------------------	--	------------------

Example: default.command: myprog.4ge arg1 arg2

default.font	Sets the default font for the application	Character string
---------------------	---	------------------

Example: default.font: Courier New-medium-r-10-

The server, port and command options are used in a script file, if the supplication is to be invoked with a script. The default resource has a number of other properties besides the ones listed above which will be discussed in the next chapters.

Setting the Font

The font property can be used for the default object as well as for any other object beginning from the application level. Thus you can set the font for a specific window, for the complete application as well as for a string or a field.

The value assigned to the font property is a character string which consists of the font characteristics separated by hyphens as follows:

```
font name-weight-slant-flags-size-special character set
```

Here flags stand for "u" - underlined and "s" - strikethrough, slant stands for "i" - italics or "r" - regular. If you do not need any of the characteristics, you should leave the place empty, but you should still include all the hyphens, e.g.:

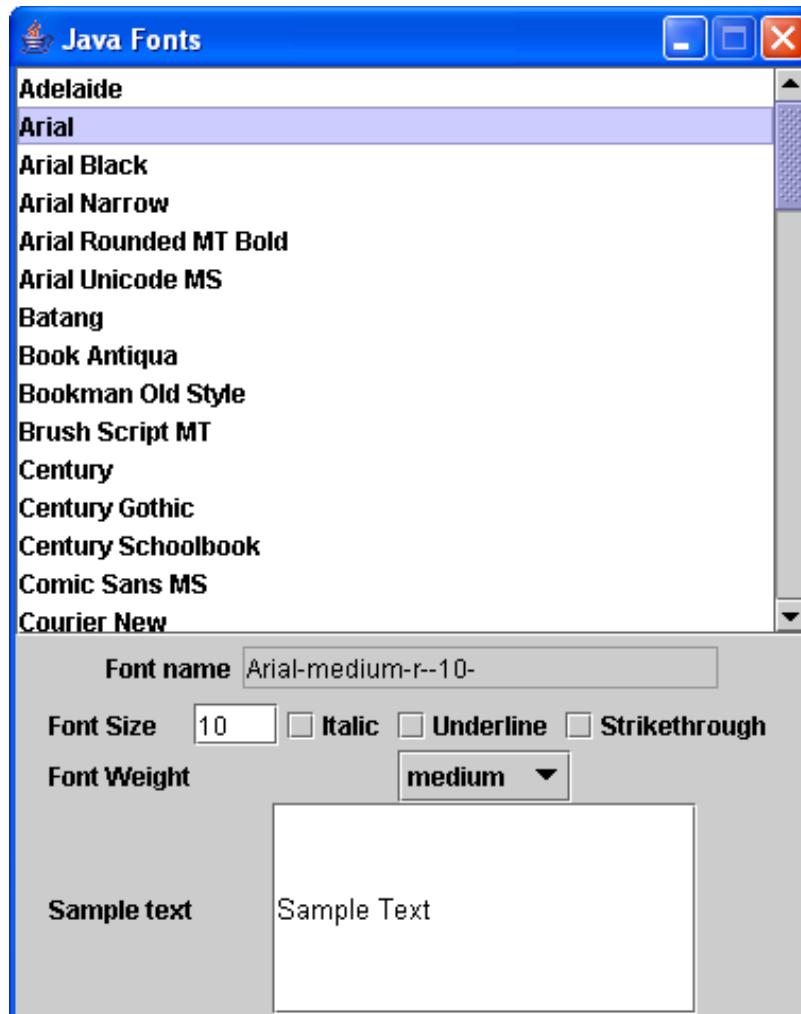
```
default.font: SansSerif-medium-i- -12-russian
```

This specifies a font of name 'SansSerif', weight 'medium', slant 'italic', no flags and a size of 12 characters and the character set 'russian'.

You can also use the font property for a specific window by specifying its address, e.g.:

```
#has the same effect as the default object
??.font: Arial-bold-i-16
# specifies font for a single window
cms.my_win1.font: Arial-bold-i-16
```

To be on the safe side and to set the fonts correctly you can use the special tool supplied by Querix and called FontList. It is located in the Tools folder of your Querix folder in the Start menu. It is shipped together with Chimera thin client, here is what it looks like:



You should select the font and its attributes and then copy the line from the "Font name" field to the font property value.

You can specify a specific font for a window, for a form or for a single field. Here are several examples:

```
#Setting the font for anything displayed to the screen
app.Screen*font: Georgia-light-r- -15-
#Setting the font for everything in the specific form
app.? .my_form*font: Lucida Sans-heavy-r- -10-
```

Colour Properties

A color property is used to specify the colour which is used to display an object. This can be approached in two ways - certain objects do not have a colour in the context of Q4GL, and assigning colours for these is more of a cosmetic issue. Other objects (e.g. text) have a Q4GL colour associated with them.

Consider, for example, the following fragment of Q4GL:



```
DISPLAY my_text
AT 20,1
ATTRIBUTES(YELLOW)
```

On a character based system, this will display the string value 'MY TEXT' in yellow. Within the GUI, this will be displayed using the RGB colour value, which is assigned to the colour yellow, because the GUI mode supports the RGB colour system. This widens the range of colours available for using in the graphical mode. However, as Querix 4GL language supports only 8 basic colours, you cannot apply the whole range of colours, if you do not use a script file for that.

Colour Specification

You can specify colour as one of the 8 colour names supported by the 4GL or in the GRB format. The colours supported by the 4GL:

- RED - red
- BLUE - blue
- GREEN - green
- MAGENTA - magenta
- CYAN - cyan
- YELLOW - yellow
- WHITE - white
- BLACK - black
- NORMAL - default

The GRB format allows you to specify any shade or hue you want. Script files use the RGB format used in HTML - a hexadecimal notation for the combination of Red, Green, and Blue colour values. These values are graphically represented by 3 pairs of two-digit numbers, starting with a # sign. For example #FF6600 specifies orange colour and #6600CC - violet.

Applying Colours

The easiest way to apply colour property is to specify 'color' keyword as the resource property. The value which can be specified for it can only be one of the 8 basic colours used in 4GL and as the result of the object will be displayed in that colour. For example, if you specify the following line in your script file, all the text throughout the program will be displayed as red:

```
*color: red
```

You can make the colour specification more exquisite. For example, you can specify what the "yellow" colour should look like when the program is run in the GUI mode. To do that you should specify the new colour for one of the default colour keywords. The colour specification in the GUI mode is more complex than in character mode. One default colour (i.e. yellow) can have up to five different options, which will be applied depending on the type of the object. For now we will only need two of them:

- foreground – specifies the foreground colour of all objects (such as displayed text)



- inactive – specifies the background colour to use for inactive objects. This includes most objects unless they are currently in use for input (i.e. the window background)

To apply colour properties use these options with the keyword specifying one of the 4GL standard colours:

```
*option.color_name: value
```

Here “option” is either foreground property or inactive property. When specifying the template, you should use the asterisk as the preceding symbol instead of the address, as the template itself does not influence the display mode until it is applied using the ‘color’ keyword. The code below applies the colour for the background and the colour for the foreground to be used instead of the standard 4GL yellow colour:

```
*foreground.yellow: #ff9900  
*inactive.yellow: #ffff99
```

The foreground colour of the objects with the yellow attribute will be orange and the background colour will be pale yellow. Now we have a modified yellow colour which defines the colour for the background and foreground. A template need not necessarily define all the possible options, you can specify only foreground colour or only background colour.

The yellow colour created in such a way can be used in two ways:

- In the 4GL code all the objects which have ATTRIBUTE (YELLOW) specification will be displayed using the newly defined yellow colour:
 - The following will result in a window having a pale yellow background, for there window area is a inactive object:

```
OPEN WINDOW win2 AT 2,2 WITH 7 ROWS, 25 COLUMNS  
ATTRIBUTE (BORDER, YELLOW)
```



- The following code will result in the orange text, because the output of the DISPLAY statement is treated as a foreground object:

```
DISPLAY "A string" AT 4,4 ATTRIBUTE (YELLOW)
```



A string

- You can apply the colour to different objects using your script file using the color option.
 - The following script option will result in a window having yellow background and orange objects:

```
app.win2*color: yellow
```

- The following script option applies the yellow colour to a single field:

```
app.win2.my_f.formonly.f001.colour: yellow
```

- Using the color keyword with the asterisk you can apply the yellow colour for the whole application:

```
*color: yellow
```

Using Non-Default Colours

You can also declare a colour pattern which name does not coincide with one of the standard 4GL colours. It is done in the same way as shown above, e.g.:

```
*foreground.my_colour: #660099  
*inactive.my_colour: #ff66ff
```

These options declare a new colour pattern called "my_colour" which consists of the violet colour for the foreground and pink for the inactive objects. The only difference between declaring the standard colours and specifying your own colour pattern is that your colour pattern cannot be used in the 4GL code. If you want to apply it, you need to use the script options, e.g.:

```
app.win2.my_f.formonly* colour: my_colour
```

Moving objects

There are many reasons to change the window layout. In this section we will discuss some methods of moving the objects around the screen. Generally speaking, we can identify three ways of moving the objects. You can:

- Move objects by a certain offset (vertical, horizontal, or both)
- Move all the objects placed on a row to another one
- Move objects to an explicit row or column



Moving Objects by a Certain Offset

If an object appears in a not appropriate place when you run the program in the GUI mode, you can write a script file to move this object to another place. For example, the whole or a part of the window content may be moved up or down to clear some space for an image or another object.

To specify the new line for one or several objects, use the *delta* commands which were discussed shortly at the beginning of this chapter. The *deltaY* and *deltaX* commands specify by how much lines and columns, respectively, the object is to be moved. If you use positive numbers (2, 5, 13), the object will be moved down or to the right; to move the object up or to the left, specify the negative numbers (-2, -5, -13). The *delta* commands are specified in the following way:

```
address.object.delta_command: integer
```

When the program evaluates delta commands in the script file, it executes the command which has the highest level of specificity. For example, if you use a wildcard to move all the objects by six columns to the right, and another command specifies that one particular object is to be moved three lines up, this one will be moved up, and the other object will be moved to the right:

```
# the top left corner of the window will be moved by 10 columns to the right
# and by 10 lines down from the position specified in the 4GL code
my_prog.my_win.deltaX: 10
my_prog.my_win.deltaY: 10

# the character string will be moved up by 3 lines
my_prog.win1.form1.my_str.deltaY: -3
```

Moving all Objects off a Particular Row

Sometimes it is necessary to move the information away from some specific lines on the screen. These can be the reserved lines, for example, the Prompt line, or the Menu line (which are the first two lines of the screen, if the application is run in Phoenix). For that the "line" property is used. This property must be followed by the number of the line on the screen, e.g. line0 refers to the first line, line1 refers to the second line. The following example demonstrates how these lines can be moved to other lines of the screen:

```
My_prog.Screen*line0: -2
My_prog.Screen*line1: -1
```

When this script is applied to the program, anything displayed to the line 0 (the first line of the screen or a window) will be moved to the last but one line of the window, and everything displayed to the line 1 (the second line of the screen or a the window) will be moved to the last line of the window.

Moving an Object to a Specific Row or Column

We have already discussed how an object can be moved by a specific number of rows or columns. However, it is also possible to move an object to a specified place of the screen. You can do it by applying new coordinates for one or several objects.

For example, if you want to move a window named *win1* to a new place, you can add the following lines to the script file:



```
# specifies the new location at the 10th line  
.win1.y: 10
```

```
#Specifies the new location at the 5th column  
.win1.x: 5
```

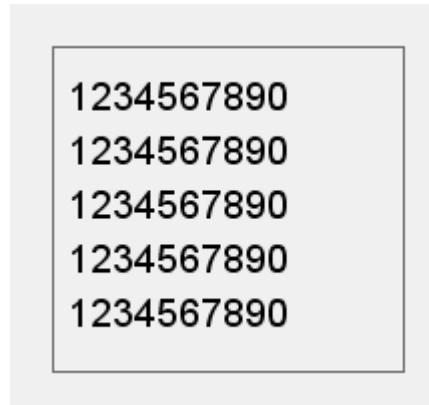
If you omit one of the coordinates when specifying the new location for the object, the omitted parameter won't be changed.

Window Style

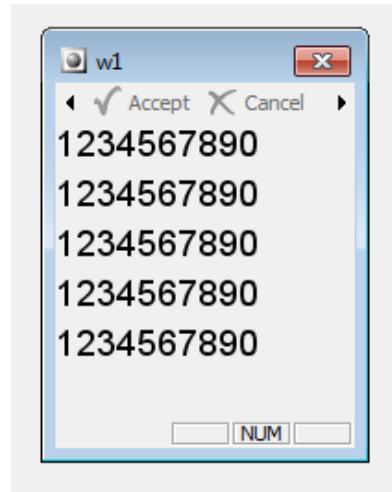
The *windowStyle* property is used to specify the rendering style for all the windows opened by the application. Q4GL provides a number of ways to set the window style, they are equally valid, and their usage depends only on the context.

There are three values you can set to the *windowStyle* property:

- *Flat* - If this value is specified, the windows will have no external frame and will have minimal decoration. The windows which have no BORDER attribute will have no visible border between them and the parent window. The windows with the BORDER attribute will have a border one column/row width. In the following example the window is opened with the *windowStyle: flat* property applied, this window has the BORDER attribute specified in the 4GL code:



- *Windowed* - each window is treated as an individual window, and is decorated individually, with no reference to other script settings specifying window rendering.



- *Mixed* - This option makes the windows that have no border be merged within the topmost window frame, and the windows which have the border re-displayed in the windowed mode.

You can use the *windowStyle* property with the *default* resource as well as with a window level resource.

```
# on an application level
?.windowStyle: windowed
# on an windows level
?.window_name.windowStyle: flat
#using the default resource
default.windowStyle: windowed
```

Templates

In script files a template is a named container which contains a number of various properties and their values. One template can contain the colour and font attributes as well as other properties. To create a template place the exclamation mark (!) at the beginning of the script line and then specify the name of the template. Then specify the property and the value. Here is the general syntax:

```
!template_name.property: value
```

To specify several properties for one template you should use several script lines declaring the template with the same name. Here is an example of a template which specifies the font and the offset:

```
!my_temp.deltaX: 10
!my_temp.deltaY: 5
!my_temp.font: Lucida Sans-medium-r-u-12-
```

Once created, a template can be assigned to any object. To assign a template specify the object and its address, then specify the "template" property and place the template name you want to use as the value. Make sure that the template name is preceded by the exclamation mark.



```
application.window.form.table.field.template: !template_name
```

The example below assigns the template to all the information displayed on the 4GL screen:

```
cms.my_win.form2.formonly.f001.template: !my_temp
```

Example

This example consists of three files: the .4gl file, the form file and the script file. They must be located in one folder and in one program. The program must be called "examples", otherwise some script options referencing the program name will not work.

The 4gl file:

```
#####
# This example illustrates the basics of the script files application
#####
MAIN

CALL strings()    -- this functions is to demonstrate possible operations
                  -- with text strings

CALL coordinates() -- this function is to demonstrate how the location
                   -- of objects can be changed

END MAIN

#####
# This function demonstrates how the texts strings can be
# influenced by script options
#####
FUNCTION strings()

OPEN WINDOW strings_window AT 2,2 WITH 20 ROWS, 60 COLUMNS
ATTRIBUTES (border)

DISPLAY "The string below is hidden" AT 1,2
# This string will not be hidden by the scripts file
DISPLAY "The string that will not be displayed" AT 2,2

#This string will be replaced by another string
DISPLAY "The string to replace" AT 3,2

# These strings will be displayed with different font strings
# applied to them
DISPLAY "The string in normal font" AT 5,2
DISPLAY "Bold Courier in my_colour" AT 6,2
DISPLAY "Italics Arial" AT 7,2
DISPLAY "Strikethrough Calibri" AT 8,2
DISPLAY "Size changed Cambria" AT 9,2
DISPLAY "Mixed parameters Tahoma in blue" AT 10,2
```



```
SLEEP 10
CLOSE WINDOW strings_window
CLEAR SCREEN
END FUNCTION

#####
# This function demonstrates how the location of the objects can
# be changed
#####
FUNCTION coordinates()

#This window will be displayed at coordinates 3, 20
OPEN WINDOW win_1 AT 10, 5 WITH 20 ROWS, 60 COLUMNS
ATTRIBUTES (BORDER)
DISPLAY "The window was located AT 10,5" AT 1,2
DISPLAY "but is displayed AT 3,30" AT 2,2

# The form field 'field' will be moved to he centre
# of the form
OPEN FORM mov_form FROM "form_script"
DISPLAY FORM mov_form
DISPLAY "Field is moved to the center" AT 15, 2
DISPLAY "The line displayed AT 16,2 is moved to the bottom" AT 16,2

CALL fgl_getkey()
CLOSE WINDOW win_1
CLEAR SCREEN

END FUNCTION
```

The Form File

The form file is called "form_script.per"

```
DATABASE formonly
SCREEN
{
  \gp-----q\g
  \g|\g form1\g          |\g
  \g|          [ f ] |\g
  \gb-----d\g
}

END

ATTRIBUTES
f = formonly.field;
```



```
END

INSTRUCTIONS_DELIMITERS " "
END
```

The Script File

The script file can have any name, which does not coincide with the name of the program:

```
#####
# Colour definition section
#####
# we define our own colour (which is violet)
*inactive.my_colour: #FF6699
*foreground.my_colour: #663399

# and define a new shade for the default colours
*inactive.yellow: #EDE275
*foreground.yellow: #7E2217

*foreground.blue: #6698FF

#####
# Script_strings section
#####
# This line sets the default font as Courier New, font size 6
default.font: Courier New-medium-r-6-

# these lines will hide the string starting with 'the string that will not'
examples.strings_window.no-form.strings.to_hide: The string that will not*
examples.strings_window.no-form.to_hide.hidden: true

# These lines will replace the string 'The string to replace'
# with another string
examples.strings_window.no-form.strings.to_replace: The string to replace
examples.strings_window.no-form.to_replace.text: This string is replaced\
by another one

# These lines will make the line starting with 'Bold' be displayed
# in bold Courier new and in a programmer-defined colour
examples.strings_window.no-form.strings.bold: Bold*
examples.strings_window.no-form.bold.font: Courier New-bold-r--12-
examples.strings_window.no-form.bold.color: my_colour

# These lines will make the line starting with 'Italics' be displayed
# in bold italics Courier new
examples.strings_window.no-form.strings.italics: Italics*
examples.strings_window.no-form.italics.font: Arial-bold-i--12-

# These lines will make the line starting with 'Strikethrough' be displayed
# in strikethrough Calibri
examples.strings_window.no-form.strings.strikethrough: Strikethrough*
examples.strings_window.no-form.strikethrough.font: Calibri-medium-r-s-12-
```



```
# These lines will make the line starting with 'Size changed' be displayed
# in Cambria with font size 9
examples.strings_window.no-form.strings.size: Size changed*
examples.strings_window.no-form.size.font: Cambria-medium-r--9

# These lines will make the line starting with 'Mix' be displayed
# in bold italics strikethrough Tahoma in blue and with font size 20
examples.strings_window.no-form.strings.alltogether: Mix*
examples.strings_window.no-form.alltogether.font: Tahoma-bold-i-us-20-
examples.strings_window.no-form.alltogether.color: blue

#####
# Script coordinates section
#####
# These lines will move the window win_1
# up and right

examples.win_1.y: 3
examples.win_1.x: 30

# the following lines move the field 'field' of the
# form mov_form 13 columns left and 5 rows up

examples.win_1.mov_form.formonly.field.deltax: -13
examples.win_1.mov_form.formonly.field.deltay: -5

# This line will remove the text string from the line 2
# of the window win_1 and put it to the last but three line
examples.win_1.mov_form.?line15: -1

# this line will make all the output performed to the
# window win_win1 be displayed in yellow defined above
?.win_1*color: yellow
```





Conditional Statements and Logical Expressions

In 4GL, like in other programming languages, a *conditional statement* is a statement that directs the computer to perform actions depending on some value. Usually this value is specified by a *Boolean expression* which can typically return only two values: *true* or *false*.

Boolean Values and Expressions

A *Boolean expression* is an expression that returns only TRUE, FALSE, and sometimes NULL value. These Boolean values are stored as integers: 1 for the TRUE value and 0 for the FALSE value.

Boolean expressions use *Boolean operators* and *relational operators*. Boolean operators are presented with logical operators AND, OR, NOT, and they combine Boolean values into Boolean expressions. Relational operators are used to compare values of non-Boolean data types in order to get a Boolean value (e.g., \leq , \neq , LIKE, etc.).

Relational Logical Operators

Relational logical operators are operators that compare values and check the relation between them. The relational operators include mathematical equality ($=$) and different types of mathematical inequalities ($<$, $>$, \neq , etc.)

The relational operators are put between their operands which may be presented by expressions or separate values. An expression created by means of relational operators is called a relational expression, or a *condition*.

The table below describes the relational logical operators and their influence on the value of Boolean expressions.

Operator name	Symbol	Returned values	
		TRUE	FALSE
<i>Equal to</i>	$=$ (==)	$3=2+1, 3==2+1$	$3=2, 3==2$
<i>Not equal to</i>	\neq (!=)	$3\neq 5, 3!=5$	$3\neq 5-2, 3!=5-2$
<i>Greater than</i>	$>$	$10>2$	$2>10$
<i>Less than</i>	$<$	$3<10$	$10<3$
<i>Greater than or equal to</i>	\geq	$6\geq=2*3, 7\geq=2*3$	$5\geq=2*3$
<i>Less than or equal to</i>	\leq	$3\leq=10-2, 3\leq=10-7$	$3\leq=1$



	<p>Note: Don't confuse the <i>equal to</i> operator with the assignment symbol used in the LET statement (=)!</p>
---	--

The relational operators can be applied to any numeric values (including Boolean values). You can also use the *equal to* (=) operator to compare character strings ("John"="Jack" → FALSE). You can't compare an INTERVAL value to a DATETIME or a DATE variable. In general, it's better to compare the values of similar data types; otherwise, you can get invalid results or errors.

Boolean Operators

The AND operator represents the *logical conjunction*. The expression created by the AND operator is evaluated as TRUE if *both* operands are TRUE, otherwise, the expression is evaluated as FALSE. Usually it combines two Boolean values. The expression below evaluates to TRUE on the whole, because each of its Boolean operands evaluates to TRUE:

```
3=1+2 AND 4=8/2 --returns TRUE because 3=1+2 is TRUE and 4=8/2 IS TRUE
```

If at least one of the operands of the AND operator returns FALSE, the whole expression evaluates to FALSE:

```
3=4 AND 4=5-1 --returns FALSE because 2=4 is FALSE though 4=5-1 is TRUE
```

The OR operator represents *logical disjunction*. The expression created by means of the OR operator is evaluated as TRUE if *one or more* operands are TRUE.

```
#The expressions below return TRUE
3=4 OR 4=5-1
3=1+2 OR 4=8/2
#The next example returns FALSE
2<1 OR 1!=1
```

The NOT operator represents *logical negation*. The NOT operator changes makes a TRUE expression be evaluated as FALSE and makes FALSE expressions be evaluated as TRUE (if expression A is true, the expression NOT A will be false).

Results of Boolean Expressions

A Boolean expression is TRUE, if it returns:

- Any DATE or DATETIME value
- An INTERVAL value, if it isn't equal to zero
- A real number, if it is not a zero
- A character string that consists of a real number, if it's not a zero.
- NULL and is used in a NULL test

A Boolean expression has the NULL value, if it returns NULL and is not used in:



- 4GL conditional statements
- Boolean comparisons
- NULL test

In all the other cases a Boolean expression is evaluated as FALSE.

The BOOLEAN Data Type

It is convenient to have an ability to store the information returned by Boolean expressions. Thus, Q4GL supports the BOOLEAN data type, which is a simple data type which can get 1, 0, and NULL values. The declaration of a BOOLEAN variable should be performed as follows:

```
DEFINE a BOOLEAN
```

```
DEFINE a BOOL
```

BOOL is a synonym of BOOLEAN.

There are three ways to assign a value to a Boolean variable. You can:

- Assign a whole number

```
LET a = 1 --sets a to TRUE
```

When assigning a value this way, you should remember that only two numbers can be used here; they are 1 and 0. No other number can be assigned to a Boolean variable. If you try to do it, the program will produce an unpredictable result.

- Assign specific keywords:

```
LET a = NULL
LET b = TRUE
LET c = FALSE
```

You can use only TRUE, FALSE, and NULL keywords in upper or lower case to assign a value to a Boolean variable. Other words used as a right-hand operand will cause an error.

If you assign the value using a quoted string or a keyword, the program transforms the value into 1, 0, or NULL before it is stored to the Boolean variable.

A Boolean variable can store values returned by Boolean expressions. Such variable can be useful when you have to refer to the result of the same expression evaluation for several times, and it is not convenient to reproduce the expression each time you need it:

```
DEFINE vbool BOOLEAN

LET vbool = 4>3 AND (5=3 OR 7<8)-- returns 1 (=TRUE)

DISPLAY vbool
```



The variables of BOOLEAN data type can be converted into most of the simple data types, and all the simple data types can be converted into a BOOLEAN value. Q4GL converts Boolean values to other data types following the rules given in the table below:

Source Data Type	Resulting Data Type	Restrictions	Resulting Values		
			TRUE	FALSE	NULL
BOOLEAN	CHAR, VARCHAR, STRING	no restrictions	"1"	"0"	NULL
	INT, SMALLINT, DECIMAL, FLOAT, SMALLFLOAT	no restrictions	1	0	NULL
	MONEY	no restrictions	\$1.00	\$0.00	NULL
	DATE, DATETIME, INTERVAL	cannot be converted	-	-	-

When Q4GL is to convert values of simple data types into Boolean values, it returns *true*, *false*, or *null* value depending on the contents of the converted value and according to the following rules:

Source Data Type	Resulting Data Type	Restrictions	Values converted into:		
			TRUE	FALSE	NULL
BOOLEAN	CHAR, VARCHAR, STRING	no restrictions	"1", "TRUE", "true"	"0", "FALSE", "false"	Any other value or NULL
	INT, SMALLINT, DECIMAL, FLOAT, SMALLFLOAT, MONEY	no restrictions	non-zero values	zero values	NULL
	DATE, DATETIME, INTERVAL	no restrictions	non-zero values	zero values	NULL

The IF Statement

The IF statement directs a program to execute one or one of two statements (or sets of statements) depending on whether the condition given as a Boolean expression is met or not.

The general syntax of the IF statement is as follows:

```
IF expression THEN statement block [ELSE statement block] END IF
```

The idea of the IF statement operation is that the program checks the Boolean expression following the IF keyword, and its value influences the choice of the executable statements.

If the Boolean expression following the IF keyword is evaluated as TRUE, the program executes the statement specified after the THEN keyword. If the Boolean expression is evaluated as FALSE, the program executes the statement following the ELSE keyword. If the ELSE keyword is omitted, no actions are taken and the program executes the statement that follows the END IF keywords.

```
DEFINE a INTEGER
```



```
...
IF a<10 THEN DISPLAY "a<10"
ELSE DISPLAY "a >= 10"
END IF
```

If an integer variable *a* is less than 10, the string "a<10" will be displayed to the screen. If the value of the variable *a* is 10 or more, the string "a >=10" will be displayed.

After the statements following THEN or ELSE keywords are executed, the program goes to the statement that follows the END IF keywords in the program code.

You can include only one condition into the IF statement and put only one statement block after the THEN or ELSE keywords, but you can nest up to 20 IF statements in one another:

```
IF age<4 THEN DISPLAY "A baby"
ELSE
    IF age<13 THEN DISPLAY "A child"
    ELSE
        IF age<18 THEN DISPLAY "A teen"
        ELSE DISPLAY "An adult"
    END IF
END IF
END IF
```

This part of a program evaluates the age of a person and defines whether he or she is a baby, a child, a teenager, or an adult, depending on the value of the *age* variable.



Note: When you use several IF statements, be attentive: the number of the END IF keywords in the end must match the number of IF keywords, otherwise a compile-time error will occur.

You can use different variables as the conditions for nested IF loops. However, if you want to check different values of one variable, it is more convenient to use the CASE statement.

Sometimes it is convenient to add some spot variables available only within the IF statement. You can do it by adding a declaration clause to the THEN or the ELSE clause. The variables specified in this way are available only within the clause they are located in. If you want both THEN and ELSE clauses to include spot variables, you have to specify two declaration clauses. If you want both THEN and ELSE clauses to use the same variables, these variables should be of local, module, or global scope:

```
DEFINE a INTEGER -- a local variable
```



```
...
IF a > 10 THEN
    DEFINE b INT -- a spot variable for the THEN clause
    LET b = a*2
    DISPLAY "B = ", b
    ELSE
        DEFINE c INT -- a spot variable for the ELSE clause
        LET c = a+10
        DISPLAY "C = ", c
END IF
```

The CASE Statement

The CASE statement is used to apply a range of conditions to statement selection. Unlike with the IF statement, you can put several conditions to the CASE statement and direct the program to perform several actions.

The syntax of the CASE statement is as follows:

```
CASE [(expression)] [Declaration clause]
      WHEN condition
            statement block
      [OTHERWISE
            statement block]
END CASE
```

The WHEN clause contains one or more executable statements which are executed when its condition is met. One CASE statement can contain several WHEN clauses.

There are two types of CASE statements:

- CASE I: the CASE keyword is **followed by an expression**. This expression specifies the value against which the conditions of the successive WHEN clauses will be checked. In such case the WHEN clauses must contain INT, SMALLINT, DECIMAL, CHAR, or VARCHAR expressions as the conditions. 4GL will perform the statement in a WHEN block, if the *condition* and *expression* return the same value and this value is not NULL.

```
CASE age
      WHEN 10 DISPLAY "A boy"
```



```
WHEN 15 DISPLAY "A man"  
WHEN 60 DISPLAY "An old man"  
END CASE
```

This type of the CASE statement is appropriate when you can check exact values of an exact variable or expression. If you don't, use the second type of the CASE statement:

- CASE II: the CASE keyword is **not followed by an expression**. The WHEN clauses may operate with different values and variables. In the second type of CASE statements, the condition of the WHEN clause must contain a Boolean expression. If this Boolean expression is evaluated as TRUE, the program executes the WHEN clause.

```
CASE  
WHEN age<10 DISPLAY "A boy"  
WHEN city = "London" DISPLAY "A Londoner"  
WHEN date_t = DATE DISPLAY "Today is ", date_t  
END CASE
```

For both CASE I and CASE II you can add an OTHERWISE clause to declare actions that are to be performed if no WHEN condition is evaluated to TRUE:

```
CASE  
WHEN age<10 DISPLAY "A child"  
WHEN age<15 DISPLAY "A teen"  
OTHERWISE DISPLAY "An adult"  
END CASE
```

You can't use Boolean expressions in CASE I statements (e.g., CASE a WHEN a<5 or CASE a WHEN <5). If you need to use Boolean expressions, use a CASE II statement (CASE WHEN a<5).

As with the IF statement, you can add a declaration clause to the CASE statement in order to specify one or several spot variables. The declaration clause of the CASE statement should appear before the first WHEN clause. The variables specified in this way are available for statements in all WHEN clauses and in the OTHERWISE clause.



```
CASE
    DEFINE ch CHAR(20)

    WHEN a > 10

        LET ch = "A is more than 10"

        DISPLAY ch at 2,2

    WHEN a < 10

        LET ch = "A is less than 10"

        DISPLAY ch at 2,2

    OTHERWISE

        LET ch = "A is equal to 10"

        DISPLAY ch at 2,2

END CASE
```

You cannot specify a declaration clause within a WHEN or an OTHERWISE clause. This will cause a compilation error.

After a WHEN clause or the OTHERWISE clause is executed, 4GL continues execution from the first statement following the END CASE keywords. When a program executes a CASE statement, it checks the conditions specified in WHEN clauses one by one, and executes the statement following the first TRUE condition. After this statement is executed, the CASE statement execution is terminated and the other WHEN clauses, even if their expressions are TRUE, are ignored.

The EXIT CASE statement

The EXIT CASE keywords are used to terminate the execution of statements located in the WHEN or OTHERWISE clause. After the EXIT CASE statement is executed, the CASE statement is terminated and all the statements between the EXIT CASE and the END CASE keywords are ignored:

```
CASE
    WHEN name_1 MATCHES "John"
        EXIT CASE

    OTHERWISE
        DISPLAY "Invalid name entered"

END CASE
```



In this example, the program checks the value of the variable *name_1*, and if it is equal to "John", the statement execution will be terminated.

The TRUE and FALSE Keywords

You can use TRUE and FALSE keywords in your Boolean expressions in the same way as you would use other Boolean values. These are built-in values. TRUE is evaluated as 1 and FALSE as 0. They can be used as operands with relational operators, e.g.:

```
LET a=1
IF a=TRUE THEN
    DISPLAY "Variable a has TRUE value"
END IF
```

The MATCHES and LIKE Operators

MATCHES and LIKE operators are used to compare character values. They check whether their left-hand operand (an expression, a variable value, or a quoted string) matches a right-hand operand, and return TRUE if it does. The left-hand operand is typically a quoted string that can include wildcard characters (characters that determine comparison options). If one of the operands has a null value, the expression itself returns NULL.

The simplified syntax of character string comparison is:

"character expression" (NOT) MATCHES/LIKE criterion

where the NOT keyword is optional, and the *criterion* is presented in a form of a character expression. Here is an example of the MATCHES operator usage:

```
IF "John" MATCHES "Jill" THEN DISPLAY "John=Jill"
ELSE DISPLAY "Values are not equal"
END IF
```

This expression will be evaluated as FALSE, because "John" and "Jill" do not match. So, this part of the source code will display: "Values are not equal".

You can use variables as operands of MATCHES and LIKE operators:

```
LET name_1 = "John"
LET name_2 = "John"
IF name_1 LIKE name_2
```



```

        THEN DISPLAY "Same names"
        ELSE DISPLAY "Values are not equal"
    END IF

```

It is also possible to compare a value to a substring of a character string. To do it, put the number of the first and the last character of the string into the square brackets following the right-hand operator:

```
IF "Joh" MATCHES name_2[1,3] THEN...
```

The MATCHES and LIKE operators are similar in functions, but they use different wildcard characters. The wildcard characters and their descriptions for the MATCHES and LIKE operator are given in the table below.

Character		DESCRIPTION	Boolean value examples	
MATCHES	LIKE		MATCHES	LIKE
*	%	An asterisk(*) or a percent sign (%) matches a string of zero and more characters	"Abcde" MATCHES "Abcde" LIKE "A%de" "A*de"	
?	_	A question mark (?) or an underscore (_) matches any single character	"Abcde" MATCHES "Abcde" LIKE "A_cd_" "A?cd?"	
[]		Square brackets match any of the characters enclosed in them (the left-hand operand must include only one character).	"A" MATCHES - "[A,b,c,d,e,f,g,h]"	
^		An initial caret put in the brackets means "except". For example, [^ajz] means that a string should match any character except a, j, and z. The left-hand operand must include only one character.	"A" MATCHES - "[^jhijkl]"	
-		A hyphen indicates a range in a collation sequence (e.g., [A-c] matches characters <i>a</i> , <i>b</i> , <i>c</i> , <i>A</i> , <i>B</i> or <i>C</i>). The left-hand operand must include only one character.	"c" MATCHES - "[a-z]"	
\	\	A backslash makes 4GL to recognize the following character as a literal one, but not as a symbol or a wildcard character. The backslash mustn't be used within the brackets that specify	"Why?" MATCHES "100%" LIKE "100\%" "WHY\?"	



the MATCHES operator range.

You can use one or several wildcard characters in one MATCHES or LIKE operand.

If you use the LIKE statement, the MATCHES wildcard characters, if any in the compared strings, will be treated as usual characters. If you want the compared strings to include a backslash, you should double it:

```
"C\\Program\\\" MATCHES "C\\Program\\\"
```

The NULL Test

When you perform a Boolean comparison, the value of the operand influences the value of the Boolean expression. If any operand has a NULL value, the expression is evaluated as FALSE. However, sometimes it is necessary to perform operations on NULL values.

The NULL test is performed to check whether a variable has the NULL value or not. You can use IS NULL and IS NOT NULL operators to perform the NULL test. If the IS NULL operand value is *null*, the expression is evaluated as TRUE. If IS NOT NULL operand value is *null*, the expression is evaluated as FALSE. Here is an example of the NULL test application in a program (a modified source code from the [The EXIT CASE](#) statement part that first checks if any value is assigned to the *name_1* variable):

```
CASE

    WHEN name_1 is not null
        IF name_1 matches "Johnny"
            THEN EXIT CASE

        ELSE
            DISPLAY "Invalid name entered"
        END IF

    WHEN name_1 is NULL display "No name value"

END CASE
```

Message Boxes

A good application should support interaction with the user in order to inform them about the program workflow process and to ask for what actions are to be taken further. Different dialog boxes are one of the means of such interaction.



A dialog box is a window of default size that appears on the screen and contains a message for a user or waits for the user input of some data or for user confirmation of some actions that the program is to perform.

Q4GL offers a set of built-in functions which invoke different kinds of default dialog boxes. Most of the dialog boxes return one or more values; to be able to operate these values, you may need to use the conditional statements, if they return more than one value.

Displaying a Message Box

The message dialog box is the only dialog box which does not return anything, thus it can be used without a conditional statement.

The *fgl_winmessage()* function is used to display a window with an interactive message box containing some text. The syntax of the function invocation is as follows:

```
CALL fgl_winmessage ("title", "text", "icon")
```

All the three arguments are character strings that specify the message box name, the text which the message box contains, and the icon that appears in the message box, respectively. The *icon* specification can be represented by "*Info*", "*exclamation*", "*question*", and "*stop*" keywords. Each of these keywords make the program add the corresponding icon to the dialog box. The icon will look as follows depending on the "*icon*" option specified:

- - the information icon
- - the question icon
- - the stop icon
- - the exclamation icon

The *text* of the message box is displayed in one line by default. To divide the message text into lines, us the *\n* symbols which indicate the current line brake and the start of the new line.

The message box is informative and the *fgl_winmessage()* function does not return any value. The message box has one button - "OK" which closes the window with the message box. After the window is closed, the 4GL code following this function is executed.

The part of the source code given below produces a message box with two lines of text and an "info" icon:

```
CALL fgl_winmessage("Winmessage test", "This is a message box\n created by  
the fgl_winmessage() function", "info")
```

You can use such message box to make the program report about the success or troubles that have happened during the execution of the program or its parts.



fgl_message_box() Function

The *fgl_message_box()* function, when activated, displays a simple message box and can contain a certain set of buttons. The only icon available for this message box and displayed by default is the question icon.

The structure of the *fgl_message_box()* is as follows:

```
fgl_message_box( "message_box_title" [ , "message_text" ] [ , button ] )
[RETURNING clause]
```

- The *message_box_title* is a character value which specifies the title of the message box.
- The second argument is a CHAR(255) value which contains text to be displayed within the message box.
- The third argument must be represented with an INTEGER value ranging from 0 to 4. Each of these values corresponds to a set of buttons that will appear at the bottom of the message box:
 - 0 - OK
 - 1 - OK, Cancel
 - 2 - Retry, Cancel
 - 3 - Yes, No, Cancel
 - 4 - Yes, No

If you don't specify any value for the third argument, the program will use the default value which is 0 and which will display only the "OK" button. You cannot customize the button names and their number - to create a customized message box you need to use a form displayed in a separate window, or to use the *fgl_winbutton()* function described below.

When the user presses one of the displayed buttons, the function returns an integer value, which is:

- 1 for "OK" or "Yes" button
- 2 for "Cancel"
- 3 for "No" and "Retry" buttons.

The following sample code calls a message box with three buttons which will appear on your screen when you launch the program:

```
MAIN
CALL fgl_message_box( "fgl_message_box",
                      "This is the text of the message", 3)
END MAIN
```

This will result in the following message box appearing on the screen:



However, if you call such message box and press a button displayed in it, the action will have no result. You have to bind the result of the `fgl_message_box()` function execution with a statement in order to make the user's choice influence the program workflow. Below is given a simple example of how it can be done:

```
MAIN

DEFINE choice INT

LET choice = fgl_message_box( "fgl_message_box" ,
                               "This is the text of the message", 3)

IF choice = 1 THEN
    DISPLAY "You pressed YES" AT 2, 2

    ELSE

        IF choice = 2 THEN
            DISPLAY "You pressed CANCEL" AT 2,2

            ELSE DISPLAY "You pressed NO" AT 2, 2

        END IF

    END IF

SLEEP 2

END MAIN
```

This program will display different character strings depending on what button is pressed by the user.

You can also use the [CALL](#) statement with the RETURNING clause to pass the value returned by the function to a variable. Therefore, the following line can replace the LET statement used in the example above:

```
CALL fgl_message_box( "fgl_message_box" ,
                      "This is the text of the message", 3) RETURNING choice
```

You can use the CALL statement with the RETURNING clause for every built-in function that returns some value to the program.



fgl_winbutton() Function

The *fgl_winbutton()* function is used to create a separate window with a message box supporting multiple choices. The peculiarity of this function is that the number and names of the buttons as well as the icon displayed are customized. The syntax of the function invocation is given below:

```
CALL fgl_winbutton ("title", "text", "default", "buttons", "icon",  
danger) [RETURNING clause]
```

- **Title** stands here for a charter string specifying the message box title.
- **Text** is a message that the message box will contain.
- **Default** is a character value specifying the default button selected.
- **Buttons** stands for one or several buttons separated by a pipe symbol (|). You can specify the button names yourself, e.g.: “Yes|No”, “Left|Right|Middle”, etc.. These can be any keywords you find appropriate for a button. The message box will contain as many buttons as you specify, but they all have standard width and are set in one line only. Thus the number of buttons determines the width of the message box.
- **Icon** specifies the name of the icon that is to be used for the message box. The *icon* specification can be represented by *info*, *exclamation*, *question*, and *stop* keywords. Each of these keywords make the program find the corresponding icons stored in the system. The icons displayed to the message box are the same as the icons displayed to the dialog message box.
- **Danger** is a SMALLINT value needed only to make the program compatible with the dynamic 4GL. We are not going to discuss this product in this manual, so the *danger* value will also be 1 in order to make the function operate.

The function returns a character value which corresponds to the name of the button pressed by the user.

As the *fgl_message_box()* function, the value returned by the *fgl_winbutton()* function should be engaged in a conditional loop. The following example demonstrates how it can be done.

```
CALL fgl_winbutton("fgl_winbutton", "This is the message text",  
"Yes", "Yes|NO|Cancel", "question", 1) RETURNING choice  
  
CASE choice  
  
WHEN "NO" display "You said NO" at 2,2  
  
WHEN "Yes" display "You said YES" at 2,2  
  
WHEN "Cancel" display "You said CANCEL" at 2,2  
  
OTHERWISE display "Something went wrong" at 2,2
```



END CASE

	<p>Note: 4GL is case-sensitive when comparing character values, so, you should print the buttons names within the <i>fgl_message_box()</i> in the same register as you note them in the following conditional statement. For example, if you specified the button as "Yes", "Yes" should be specified in a conditional statement, but not "YES" or "yes".</p>
---	--

The message box invoked by the code above will look as follows. As you can see the "Yes" button is highlighted as the default button:



***fgl_winquestion()* Function**

The *fgl_winquestion()* function is used to display a new window with an interactive message box containing a combination of buttons. The sets of buttons available cannot be customized, whereas the icon can.

The syntax of the function invocation is:

```
fgl_winquestion ("title", "text", "default", "buttons",
"icon", danger) [RETURNING]
```

All the arguments of the function are similar to those of the [*fgl_winbutton\(\)*](#) function. They are all necessary, no one of them can be omitted during the function invocation.

The only difference is the *button* argument. Q4GL provides the following sets of buttons for the *fgl_winquestion()* function:

- OK
- OK|Cancel
- Yes|No
- Yes|No|Cancel
- Retry|Cancel
- Abort|Retry|Ignore

You can specify only one of these sets for the message box created by the *fgl_winquestion()* function. The set of the buttons should be taken in parentheses.

The following example demonstrates how the *fgl_winquestion()* function can be used in a program and linked to a conditional statement:

```
DEFINE choice VARCHAR(10)
```



```
LET choice = fgl_winquestion("fgl_winquestion",
                               "This is the text of the message",
                               "Ignore", "Abort|Retry|Ignore",
                               "question",1)
CASE choice
  WHEN "Abort" display "You said ABORT" at 2,2
  WHEN "Retry" display "You said RETRY" at 2,2
  WHEN "Ignore" display "You said IGNORE" at 2,2
  OTHERWISE display "Something went wrong" at 2,2
END CASE

SLEEP 2
```

Here is what the message box invoked by the function described above looks like:



	Note: Remember, that the Q4GL is case sensitive when evaluating the value returned by the <i>fgl_winquestion()</i> function.
--	---

Example

This example illustrates the usage of the conditional statements. It also shows how the default dialog boxes can be effectively used with conditionals to make the conditionals depend on the user input.

```
#####
# This is an example of IF and CASE statements as well as of logical
# operations
#####
```

DEFINE



```

m_char      CHAR(80),
m_integer  INTEGER

MAIN

DEFINE
    v_integer   INTEGER,
    v_smallint  SMALLINT,
    v2_char     CHAR(20),
    answer      CHAR(5) -- This variable will store a CHAR value
                        -- returned by the dialog functions

DISPLAY "Relational <,<=,=,<>,>=,> and logical AND,OR,NOT operators in use:"
AT 1,2 ATTRIBUTE(GREEN, BOLD)
SLEEP 2
DISPLAY "TRUE:"                      FALSE:                      NULL: "
AT 2,2 ATTRIBUTE(YELLOW, BOLD)
LET v_integer = NULL

# These are relational operators

LET v_smallint = (2 < 5)-- less than operator returns TRUE (1)
DISPLAY "(2 < 5)"                  = ",v_smallint AT 3,2
LET v_smallint = (2 <= 5)-- less than or equal to operator returns
                          -- TRUE (1)
DISPLAY "(2 <= 5)"                  = ",v_smallint AT 4,2
LET v_smallint = (2 <> 5)-- not equal to operator returns TRUE (1)
DISPLAY "(2 <> 5)"                  = ",v_smallint AT 5,2
LET v_smallint = (2 > 5)-- greater than operator returns FALSE (0)
DISPLAY "(2 > 5)"                  = ",v_smallint AT 3,29
LET v_smallint = (2 >= 5)-- greater than or equal to operator
                          -- returns FALSE (0)
DISPLAY "(2 >= 5)"                  = ",v_smallint AT 4,29
LET v_smallint = (2 = 5)-- equal to operator returns FALSE (0)
DISPLAY "(2 = 5)"                  = ",v_smallint AT 5,29
LET v_smallint = (2 < v_integer)-- returns NULL, because v_integer is NULL
DISPLAY "(2 < NULL)"                = ",v_smallint AT 3,57
LET v_smallint = (5 = v_integer)-- returns NULL, because v_integer is NULL
DISPLAY "(5 = NULL)"                = ",v_smallint AT 4,57
LET v_smallint = (5 <> v_integer)--returns NULL,because v_integer is NULL
DISPLAY "(5 <>NULL)"               = ",v_smallint AT 5,57

# These are logical operators
# TRUE and FALSE keywords represent built-in values: TRUE = 1, FALSE = 0

LET v_smallint = (TRUE AND TRUE)          -- returns TRUE (1)
DISPLAY "(TRUE AND TRUE)" = ",v_smallint AT 6,2
LET v_smallint = (TRUE OR TRUE)           -- returns TRUE (1)
DISPLAY "(TRUE OR TRUE)" = ",v_smallint AT 7,2
LET v_smallint = (TRUE OR FALSE)          -- returns TRUE (1)
DISPLAY "(TRUE OR FALSE)" = ",v_smallint AT 8,2
LET v_smallint = (TRUE OR v_integer)-- returns TRUE (1) you cannot use
                                    -- NULL keyword in this case for
                                    -- it will be treated as a variable

```



```

DISPLAY "(TRUE OR NULL) = ",v_smallint AT 9,2
LET v_smallint = (v_integer OR TRUE)                                -- returns TRUE (1)
DISPLAY "(NULL OR TRUE) = ",v_smallint AT 10,2
LET v_smallint = NOT(FALSE)                                         -- returns TRUE (1)
DISPLAY "NOT (FALSE) = ",v_smallint AT 11,2

LET v_smallint = (TRUE AND FALSE)                                    -- returns FALSE (0)
DISPLAY "(TRUE AND FALSE) = ",v_smallint AT 6,29
LET v_smallint = (FALSE AND FALSE)                                   -- returns FALSE (0)
DISPLAY "(FALSE AND FALSE) = ",v_smallint AT 7,29
LET v_smallint = (FALSE AND v_integer)                                -- returns FALSE (0)
DISPLAY "(FALSE AND NULL) = ",v_smallint AT 8,29
LET v_smallint = (v_integer AND FALSE)                                -- returns FALSE (0)
DISPLAY "(NULL AND FALSE) = ",v_smallint AT 9,29
LET v_smallint = (FALSE OR FALSE)                                     -- returns FALSE (0)
DISPLAY "(FALSE OR FALSE) = ",v_smallint AT 10,29
LET v_smallint = NOT(TRUE)                                           -- returns FALSE (0)
DISPLAY "NOT (TRUE) = ",v_smallint AT 11,29

LET v_smallint = (TRUE AND v_integer)                                 -- returns NULL
DISPLAY "(TRUE AND NULL) = ",v_smallint AT 6,57
LET v_smallint = (v_integer AND TRUE)                                -- returns NULL
DISPLAY "(NULL AND TRUE) = ",v_smallint AT 7,57
LET v_smallint = (v_integer AND v_integer)                            -- returns NULL
DISPLAY "(NULL AND NULL) = ",v_smallint AT 8,57
LET v_smallint = (FALSE OR v_integer)                                 -- returns NULL
DISPLAY "(FALSE OR NULL) = ",v_smallint AT 9,57
LET v_smallint = (v_integer OR v_integer)                            -- returns NULL
DISPLAY "(NULL OR NULL) = ",v_smallint AT 10,57
LET v_smallint = NOT(v_integer)                                      -- returns NULL
DISPLAY "NOT (NULL) = ",v_smallint AT 11,57

SLEEP 15
DISPLAY "Conditional statement IF:" AT 12,2 ATTRIBUTE(GREEN, BOLD)
# IF statement executes only if its condition on the whole evaluates to TRUE,
# regardless to what evaluate individual the operands of the condition:

IF (2 < 5) THEN                                              -- expression (2<5) is TRUE so the IF
    -- statement is executed
DEFINE booll BOOLEAN -- We define a variable of the BOOLEAN data type
    -- after keyword THEN. It is a spot variable
    -- which is only visible within the if-block
    -- where it was declared.
LET booll = "true" -- We assign a value to booll in one of the three
    -- possible ways - by using a quoted character string

DISPLAY "booll is ", booll,"(TRUE)" AT 13,2
END IF

# To make sure that the booll variable is not accessible by any code outside
# the if-block where it was declared, you can uncomment the following
# statement. In this case the compiler will produce an error.
-- DISPLAY "booll is ", booll,"(TRUE)" AT 13, 2

```



```
IF (1 > 2) THEN -- expression (1 > 2) is FALSE

DISPLAY "1 is greater than 2" AT 12,55 -- This IF statement will not produce
                                         -- any output because the condition
                                         -- evaluates to FALSE and no ELSE
                                         -- clause is included.

END IF

IF (5 <> 5)-- expression (5<>5) is FALSE

THEN -- the THEN clause can be empty

ELSE DISPLAY "IF(5 <> 5) is FALSE" AT 13,29 -- the ELSE clause is executed
END IF

IF (5 = v_integer) THEN -- expression (5 = NULL) is FALSE.
DISPLAY "IF(5 = NULL) is TRUE " AT 13,57
ELSE -- the ELSE clause is executed
DISPLAY "IF(5 = NULL) is NULL" AT 13,57
END IF
SLEEP 10

DISPLAY "Conditional statement CASE is illustrated by message boxes" AT 14,2
ATTRIBUTE(GREEN, BOLD)

# We call the fgl_winbutton()function to create a dialog box with three
# buttons. Pressing one of these buttons will send a corresponding CHAR value
# to the variable answer specified in the RETURNING clause.

CALL fgl_winbutton ("CASE 1",           -- title of the dialog box
                    "Choose a value", -- text of the message displayed
                    "2",              -- the button selected by default
                    "2|0|5",          -- buttons
                    "question",       -- icon
                    1)                -- danger parameter
RETURNING answer

# The first type of the CASE statement (with an expression)
# The WHEN clause which corresponds to the pressed key will be executed

CASE answer
WHEN 5
    # This function does not return anything
    # It is used for displaying information
    CALL fgl_winnmessage("CASE results",           -- title
                         "Button 5 was pressed", -- message
                         "info")                 -- icon
WHEN 0
    CALL fgl_winnmessage("CASE results", "Button 0 was pressed", "info")
WHEN 2
    CALL fgl_winnmessage("CASE results", "Button 2 was pressed", "info")

#if you close the dialog box pressing close button (x)
```



```
# this clause is executed
OTHERWISE
    CALL fgl_winmessage("CASE results", "The dialog was closed", "info")

END CASE

# We assign the value returned by a dialog box to a variable
# so that we could execute the following code conditionally
LET answer = fgl_winquestion ("CASE 2", -- title
                               "Start CASE 2 execution?", --message
                               "Yes", -- default button
                               "Yes|No", -- buttons
                               "question", -- icon
                               1) -- danger

IF answer = "Yes" THEN
    # This function executes the CASE statement
    CALL case_2()
ELSE
    # This is a nested IF statement
    IF answer = "No" THEN -- it is executed, if you press No
        CALL fgl_message_box("CASE 2 execution", -- title
                             "Are you sure you want to skip", -- message
                             2) -- keys Retry|Cancel
        RETURNING answer
    IF answer = "2" THEN -- executed, if Cancel is pressed
        CALL fgl_winmessage("CASE 2 execution",
                            "You've chosen not to execute \n the second CASE statement",
                            "exclamation")
    ELSE -- executed if Retry is pressed
        CALL fgl_winmessage("CASE 2 execution",
                            "The second CASE statement will execute", "info")
        # This function executes the CASE statement
        CALL case_2()
    END IF
END IF
SLEEP 5
# Function func_rec() calls itself recursively for three times due to the
# CASE statement in the function body, see the FUNCTION block below

LET m_char = " "
LET m_integer = 0
CALL func_rec()
DISPLAY "Function has been called ",m_integer, " times." AT 15,27
SLEEP 2

DISPLAY "Operators IS NULL and IS NOT NULL:" AT 16,2 ATTRIBUTE(GREEN, BOLD)

# You can compare a value with NULL in conditional statements only using IS
# NULL or IS NOT NULL operators. Conditional operators are invalid in such
# context
```



```

IF v_integer IS NULL -- TRUE condition
THEN DISPLAY "IF(NULL IS NULL) TRUE" AT 17,2
ELSE DISPLAY "IF(NULL IS NOT NULL) FALSE" AT 17,2
END IF

IF v_integer IS NOT NULL -- FALSE condition
THEN DISPLAY "IF(NULL IS NULL) TRUE" AT 17,29
ELSE DISPLAY "IF(NULL IS NOT NULL) FALSE" AT 17,29
END IF
SLEEP 10
# Below the character strings are matched using the wildcard symbols allowed
# for MATCHES and LIKE operators
DISPLAY "Operator MATCHES:" AT 18,2 ATTRIBUTE(GREEN, BOLD)

LET v2_char = "abc123d"

# * stands for any number of characters
IF v2_char MATCHES "*c12*" THEN
DISPLAY '"abc123d" MATCHES "*c12*"' AT 19,2 END IF

# ? matches any single character
IF v2_char MATCHES "a?c?23d" THEN
DISPLAY '"abc123d" MATCHES "a?c?23d"' AT 19,45 END IF

# [ ] - matches any of the enclosed characters
IF v2_char MATCHES "ab[cCx]1[952]3d" THEN
DISPLAY '"abc123d" MATCHES "ab[cCx]1[952]3d"' AT 20,2 END IF

# - matches any character from the range [first-last].
IF v2_char MATCHES "a[a-z]c12[1-9]d" THEN
DISPLAY '"abc123d" MATCHES "a[a-z]c12[1-9]d"' AT 20,45 END IF

# ^ - matches any characters except the listed ones.
IF v2_char MATCHES "ab[^abde]12[^24]d" THEN
DISPLAY '"abc123d" MATCHES "ab[^abde]12[^24]d"' AT 21,2 END IF

LET v2_char = "a*c1?3d"
# \ specifies that the following character should be treated as a literal
IF v2_char MATCHES "a\*c1\?3d" THEN
DISPLAY '"abc123d" MATCHES "a\*\c1\?\3d"' AT 21,45 END IF
SLEEP 10
DISPLAY "Operator LIKE:" AT 22,2 ATTRIBUTE(GREEN, BOLD)

LET v2_char = "abc123d"

# % matches zero or more characters
IF v2_char LIKE "%c12%" THEN
DISPLAY '"abc123d" LIKE "%c12%"' AT 23,2 END IF

# _ matches any single character
IF v2_char LIKE "a_c_23d" THEN
DISPLAY '"abc123d" LIKE "a_c_23d"' AT 23,45 END IF

```



```
# \ specifies that the following character should be treated as a literal
LET v2_char = "a%c1_3d"
IF v2_char LIKE "a%\c1\_3d" THEN
DISPLAY 'abc123d' LIKE "a\\%\c1\\_3d" AT 24,2 END IF
SLEEP 10
END MAIN

#####
# The second type of the CASE statement is located
# in a separate function, which is called
# depending on the button clicked in a dialog box
#####

FUNCTION case_2()
DEFINE v_integer INTEGER
DEFINE bool2 BOOLEAN

LET bool2 = fgl_winbutton ("Setting BOOLEAN value",
                           "Set the value either to TRUE (1) \n or to FALSE (0)",
                           "1", "1|0", "exclamation", 1)

# the second type of the CASE statement (without the expression)
# It gets the value of the variable used in the WHEN clauses
# from the dialog box above
CASE
  DEFINE boolval VARCHAR(20)
  WHEN bool2 = TRUE
    LET boolval = "TRUE"
    DISPLAY "bool2 is ", bool2, 2 SPACE, boolval AT 15,2
  WHEN bool2 = FALSE
    LET boolval = "FALSE"
    DISPLAY "bool2 is ", bool2, 2 SPACE, boolval AT 15,2
  OTHERWISE -- executed if the dialog is closed
    LET boolval = "(NULL)"
    DISPLAY "bool2 is ", bool2 CLIPPED, 2 SPACE, boolval AT 15,2
END CASE
END FUNCTION

#####
# The recursive function func_rec() without arguments
# It calls itself recursively for three times and uses the module variables
# m_integer and m_char
#####
FUNCTION func_rec()

# Each of the WHEN conditions becomes true, because the m_integer changes its
# value after each function call

LET m_integer = m_integer + 1
LET m_char = "Call # ", m_integer

CASE m_integer
  WHEN 1           -- true after the initial function call
    CALL fgl_winmessage("Recursive function call",
                         m_char, "info")
```



```
CALL func_rec() -- calls func_rec() once more which results
-- in m_integer = 2

WHEN 2           -- true after the second recursive call
--(after WHEN 1 clause is executed)
CALL fgl_winmessage("Recursive function call",
                     m_char, "info")
CALL func_rec() -- calls func_rec() for the third time results
-- in m_integer = 3

WHEN 3           -- true after third recursive call
-- (after WHEN 2 clause is executed)
CALL fgl_winmessage("Recursive function call",
                     m_char, "info")
CALL func_rec() -- calls func_rec() for the fourth time results
-- in m_integer = 4

OTHERWISE        -- this condition becomes TRUE after the function is
-- called for the fourth time
-- (after WHEN 3 clause is executed)

      RETURN
END CASE

END FUNCTION
```



Program Workflow Control

A program is usually a complex structure that must lead to a certain result. There may be several ways to obtain the result, and the program code may describe some of these ways in order for the program to operate successfully. That's why some statements may prove to be useless during one session, and some may be necessary to execute for several times. Moreover, the user must be able to influence the program execution, e.g. to terminate it early.

Therefore, the program workflow control is an important part of the program development. There are several ways to control the program workflow, and in this chapter we will discuss the most popular ones.

The PROMPT Statement

The PROMPT statement is used to receive some values from a user. However, it often plays an important role in the program workflow control, whereas the input from a user is generally received using screen forms and not the PROMPT statement.

The simplest case of the PROMPT statement has the following syntax:

```
PROMPT "Character string" FOR variable
```

The "character string" represents a message displayed before the area for input. It usually tells the user what they should enter. It can contain only whitespaces, then no message will be displayed, but the user will still be able to enter values. The *variable* is a variable of any simple data type which is used to store value entered by a user. This variable is also called a *receiving* variable. The value of this variable can be used throughout the program, or it can be ignored in cases when the aim of the PROMPT statement is to ensure that the user has seen the message and 4GL can continue the execution of the program.

The PROMPT statement is often used to receive some response from the user for the program to know how to behave. In this sense it can be combined with other program workflow controlling statements (i.e. EXIT PROGRAM) to influence the program behaviour. For example it is widely used in the following situation:

```
LABEL beginning:  
PROMPT "Do you want to exit the program? (y/n) " FOR CHAR yes_no  
IF yes_no MATCHES "[Yy]" THEN  
    EXIT PROGRAM  
ELSE  
    DISPLAY "The program will continue its execution." AT 2,2  
    GOTO beginning  
END IF
```

The SLEEP Statement

We have already discussed some aspects of the [SLEEP](#) statement. Here we offer a more profound description of this statement. The SLEEP statement suspends the program execution. The length of the



pause is defined by an integer expression that follows the SLEEP keyword and specifies the number of seconds by which the program execution is suspended. Nothing can happen within the program for the specified period of time, the statements following the SLEEP statement will be executed only after the pause ends.

This statement may be useful, if you want to make screen display visible long enough for the user to see and read the displayed results before proceeding with something else:

```
DISPLAY "The program is executed" AT 3,2
SLEEP 3
DISPLAY "3 seconds have passed" AT 3,2
```

This part of the source code will display the first character string to the screen and keep it there for three seconds. Only after that the second message will appear at the same place erasing the initial message. Without the SLEEP statement you would have been able to see only the second message.

Fgl_getkey() Function

The *fgl_getkey()* function returns the integer code of a key pressed by the user. This function does not accept any arguments, the syntax of its invocation is:

```
fgl_getkey()
```

For now the value returned by this function is of no use to us, but it can be effectively used to suspend the program execution. When 4GL encounters this function, it waits for a keystroke to be performed. Thus the application does not have to be in the input mode for the user to be able to press a key. Then it returns the value of the key pressed. This is more convenient than using the SLEEP statement, while the program remains suspended for the time span defined by the user and not by the programmer.

The GOTO and LABEL Statements

In some cases an ability to pass the program control to a certain place of the program code may prove to be useful. If you want to do this, you must use the LABEL and the GOTO statements. Here is their syntax:

```
LABEL label_name:
GOTO [ :] label_name
```

The LABEL statement is used to identify the place in the program to where the program execution should be passed. The GOTO statement passes the program execution to the labelled place, which can be below or above the GOTO statement. An example of the GOTO and LABEL statements usage may look as follows:

```
MAIN
    DEFINE i INTEGER
    ...
LABEL l_sum:
    LET i=i+1
    DISPLAY "i=", i-1, 5 space, " i+1=", i
    SLEEP 1

CASE
```



```
WHEN i<15
    GOTO: l_sum
WHEN i=15
    GOTO disp_sum
OTHERWISE exit case
END CASE
...
...
LABEL disp_sum:
    DISPLAY "i =", i
...
...
END MAIN
```

This part of source code checks the value of the variable *a* by means of the CASE statement. If it is less than 15, the program control is returned to the place labelled as *l_sum*. 4GL adds 1 to the value of the variable *a*, and displays it. If the value of the variable *a* is 15, the program control jumps to the place labelled as *disp_sum*, ignoring all the statements between the END CASE and the LABEL keywords. If the variable *a* has another value (which is more than 15), the CASE statement is terminated and the program continues its execution from the first statement after the END CASE keywords.



Note: If the program control hasn't been passed to a labelled place, the statements following the label will be executed in the default order (after the previous statements are executed).

The GOTO and LABEL statements in the example above imitate the behaviour of a primitive conditional loop. The statements used to create the conditional loops are discussed in the next chapter. It is generally considered that the GOTO and LABEL statements should be replaced by the conditional loops where possible. You can also use such statements as CASE, IF, CALL, and others to perform program workflow control. The overuse of labels may lead to confusions and unreadable source code.

You should follow several rules, when you use the LABEL and GOTO statements:

- The GOTO and LABEL statements can't be located in different program blocks
- The label name must be unique within the program block where it is used.
- You must specify the same label name in both GOTO and LABEL statements to perform valid control transfer.
- The GOTO statement may be written as GO TO. These two ways of statement specification are absolutely equal.
- The LABEL name must be followed by a colon, but the colon is optional for the GOTO statement:

```
LABEL my_lab:
GOTO my_lab
GO TO: my_lab
```



The EXIT PROGRAM Statement

The EXIT statements are used to interrupt the execution of a control structure, for example, a conditional statement (EXIT CASE), or to terminate the program itself.

The EXIT PROGRAM statement is used to terminate a program early, before 4GL encounters the END MAIN keywords. In such case all the statements between the EXIT PROGRAM and END MAIN statements are skipped and the program is immediately terminated.

```
MAIN
...
...
IF a IS NULL THEN EXIT PROGRAM
    ELSE ...
...
...
END MAIN
```

If the EXIT PROGRAM statement is located outside a conditional statement or any other limiting structure, the statements following it will never be executed. Thus this statement makes sense mainly when a condition is applied.

The DEFER Statement

4GL gives the user the possibility to interrupt the program execution manually. Thus, it supports the Interrupt key which is CONTROL-C, and the Quit key which is CONTROL-\. The Quit and the Interrupt commands are generally synonymous and cause the program termination the moment they are pressed.

The DEFER statement is used to prevent the program interruption when the Interrupt or Quit keys are pressed. A program can have only one DEFER INTERRUPT and DEFER QUIT statement, and they must be located only in the MAIN program block. After the DEFER statements are executed, they remain in effect for the whole period of the program execution.

The idea of the DEFER statement operation is that when it is executed, a corresponding built-in 4GL variable gets a non-zero value. When the DEFER INTERRUPT or DEFER QUIT statements are in effect and either Interrupt or Quit key is pressed, the built-in variables *int_flag* and *quit_flag*, respectively, are set to TRUE. You can change the values of variables *int_flag* and *quit_flag* within your source code, they can have assigned values as any other variables, but they need not be declared.

Setting Options for Program Termination

There is one more way to specify some program actions for the case of program termination. You can do it by adding some clauses to the OPTIONS statement.

OPTIONS ON CLOSE APPLICATION

The user may want to terminate the program execution in the GUI mode by pressing the Close button (The "x" button in the top right corner of the application window). The OPTIONS ON CLOSE APPLICATION statement specifies the actions that the program must take in this case. There are four possible actions for the OPTIONS ON CLOSE APPLICATION statement:

```
OPTIONS ON CLOSE APPLICATION STOP
OPTIONS ON CLOSE APPLICATION CALL Function
```



- The STOP statement is used to terminate the program. This is the default behaviour which occurs when the Close button is clicked, so such OPTION statement is used to return to the default behaviour after some other behaviour has been specified.
- The CALL statement is used to invoke a programmer-defined or an in-built function and to pass the program control to this function. You cannot pass values to it, so you must not use the parentheses after the function name:

OPTIONS ON HANGUP|TERMINATE SIGNAL CALL

The ON HANGUP SIGNAL CALL and ON TERMINATE SIGNAL CALL options are used to specify the function which is to be invoked when the connection with the client is closed or terminated, respectively. The syntax of the options specification is:

```
OPTIONS ON HANGUP SIGNAL CALL function
```

```
OPTIONS ON TERMINATE SIGNAL CALL function
```

You can specify the OPTIONS clause in any part of your source code, and it will have the effect until the program encounters another OPTIONS clause for the same condition.

Titlebar Options

Earlier, we have discussed how you can change the effect of the Close button being clicked. The Close button is not the only button on the title bar; there are two other buttons - "Maximize" and "Minimize" buttons which allow you to change the size of the window manually, by clicking them. Q4GL provides a set of script options that can enable and disable these buttons thus influencing the possible range of user's influence on the program execution. You must specify these script options in a script file.

Deactivating the Close Button

This script option is used to enable or disable the close button on the application window. When the close button is enabled, it can be pressed to terminate the program or to make the program perform the actions specified in the OPTIONS ON CLOSE APPLICATION statement.

The syntax of the option specification is as follows:

```
application_name.window_name.titlebar_close: <Boolean value>
```

To disable the Close button, specify the *Boolean value* as *false*. If you want to enable the Close button, set the *Boolean value* to *true*.

Deactivating the Maximize and Minimize Buttons

You can use similar options to enable and disable the minimize and maximize buttons. The options syntax is the same as the syntax of the *titlebar_close* option:

```
application_name.window_name.titlebar_minimize: <Boolean value>
```



```
application_name.window_name.titlebar_maximize: <Boolean value>
```

You should also use *o* or *false* keyword to disable the specified button and *true* keyword to enable it.

Example

The example below illustrates the discussed issues. Some input from a user will be needed, such as pressing the Quit key or the Interrupt key, etc.

```
#####
# This example illustrates some statements which allow you to suspend the
# execution of a program, terminate a program or prevent its termination
# by a user
#####

MAIN

DEFINE c CHAR

# The Interrupt and Quit keys work properly mainly in character
# because the GUI mode was designed to behave like modern GUI applications
# so the Close(X) button can be used instead. It is handled using scripts
# and the ON CLOSE APPLICATION option.
IF fgl_fglgui()= 1 THEN
    CALL fgl_winmessage("Interrupt keys",
        "The Interrupt and Quit keys don't work in GUI mode.", "info")
    EXIT PROGRAM
END IF

DEFER INTERRUPT -- this statement prevents program termination if
                  -- CONTROL-C (INTERRUPT) key is pressed
                  -- instead it sets int_flag to 1 when this key is pressed

DEFER QUIT      -- this statement prevents program termination if
                  -- CONTROL-\ (QUIT) key is pressed
                  -- instead it sets quit_flag to 1 when this key is pressed

OPTIONS

ON CLOSE APPLICATION CALL func_close -- If the user presses the X button
                                         -- in the upper right-hand corner of
                                         -- the window, the func_close will
                                         -- be called. This will be in effect
                                         -- until another OPTIONS ON CLOSE
                                         -- APPLICATION statement is
                                         -- encountered. Notice that there
                                         -- are no parentheses following
                                         -- the function name

OPEN WINDOW win1 AT 2,2 WITH 20 ROWS, 75 COLUMNS ATTRIBUTE (BORDER)

DISPLAY "The Quit and Interrupt keys are used to terminate the program early."
AT 4,2 ATTRIBUTE(GREEN, BOLD)
```



```
DISPLAY "But the DEFER statement allows us to change the program behaviour"
AT 5,2 ATTRIBUTE(GREEN, BOLD)

LABEL lab1:          -- Here we put a label for the program to return to this
                     -- place when it encounters GOTO lab1 statement

LET int_flag = 0 -- you should set int_flag and quit_flag to their
                  -- initial values (0) before using them

LET quit_flag = 0 -- in order not to get confused in the program logic
                  -- after using them for several times

# A simple PROMPT statement. You need to press RETURN after entering a value
# You can enter any value, but the program will return you to this prompt
# statement unless you press the Interrupt or Quit key.
# These keys will not be entered as a value and you do not need to press
# RETURN afterwards
PROMPT "Please, press the Interrupt key (CONTROL-C) or ",
       "Quit key (CONTROL-\\")

FOR c
ON KEY (F1)
GOTO lab1
END PROMPT

CASE
WHEN int_flag = 1 -- When CONTROL-C (which is the INTERRUPT key) is
                  -- pressed, the built-in global variable
                  -- int_flag is set to 1 (TRUE)
DISPLAY "CONTROL-C (INTERRUPT) was pressed. The program will exit in 3",
        " seconds." AT 7,2
SLEEP 3           -- the program execution is suspended for 3 seconds
EXIT PROGRAM     -- 4GL terminates the program and exits the MAIN block

WHEN quit_flag = 1 -- When CONTROL-\ (which is the QUIT key) is pressed,
                  -- the built-in global variable quit_flag is set to 1
DISPLAY "CONTROL-\\"(QUIT)was pressed. The program will exit in three",
        " seconds." AT 7,2
SLEEP 3
EXIT PROGRAM

OTHERWISE
DISPLAY "The value entered is not CONTROL-C or CONTROL-\\",
        " returning to PROMPT!" AT 7,2 ATTRIBUTE(YELLOW, BOLD)
GOTO lab1         -- the program control is transferred to LABEL lab1:, if
                  -- you enter any value into PROMPT field
                  -- instead of pressing the Quit or Interrupt key
END CASE

CLOSE WINDOW win1

END MAIN

FUNCTION func_close() -- Function to be called when the user presses the
                      -- close button in the upper right-hand corner
```



```
CALL fgl_winmessage ("Close request", "Do you really want to quit ",  
                     "the application?", "question")  
CLOSE WINDOW win1  
EXIT PROGRAM  
END FUNCTION
```

The Script File

And these are the contents of the script file with the titlebar options. The script options do not depend on the program name.

```
# The first option enables the close button, otherwise the  
# ON CLOSE APPLICATION option will not work  
?.win1.titlebar_close: true  
# The second one disables the maximize button  
?.win1.titlebar_maximize: false  
  
# And the third one disables the minimize button  
  
?.win1.titlebar_minimize: false
```



Conditional Loops

Conditional loops are repetitive control structures that make a program repeat a set of statements for several times until a Boolean expression specified in them is TRUE. The most common conditional loops are FOR and WHILE loops.

FOR loop

The FOR loop is used to execute a set of statements (a logical block) a definite number of times. The loop starts with the FOR and ends with the END FOR keywords. Use this method when you know how many times the statements should be iterated.

The number of the loop iterations is specified by means of a counter – an INTEGER or a SMALLINT variable that serves as an index for the statement block. It is increased by the number specified by the increment of the STEP clause each time the loop is iterated.

The general syntax of the FOR statement is:

```
FOR counter = min TO max [STEP increment]  
[Declaration clause]  
  
...  
...  
... }      logical block  
  
END FOR
```

The TO Clause

The TO clause includes the counter and its *minimum* and *maximum* values. The maximum number of loop iterations is usually determined by the difference between the *first* and the *last* values of the counter plus one, because the loop is iterated from *min* to *max* including both minimum and maximum values. E.g. FOR i = 2 TO 5 will cause the FOR loop to be iterated four times, because 5-2+1=4, and the loop is iterated when i=2, i=3, i=4, and i=5.

The *first* value of the counter is set when the FOR loop is executed for the first time, and is changed before each following execution of the logical block.

When the *first* counter value becomes equal to or more than the *last* one, the FOR loop is terminated, and the program control is passed to the statement following the END FOR keywords.

Here is an example of a simple FOR loop which displays the value of its counter:

```
DEFINE i INT -- counter  
FOR i = 1 to 10
```



```
DISPLAY "i=",i CLIPPED AT i, 2
```

```
END FOR
```

This part of source code will produce 10 lines which will be "i=1", "i=2", "i=3"..."i=10". The DISPLAY statement will be displayed at lines corresponding to the /value.

The minimum and maximum values can be integers either positive or negative. However, make sure that the minimum value is smaller than the maximum one. In case of a negative counter the following is true, if the STEP clause is absent:

```
FOR i = 25 TO 50 -- a valid TO clause
FOR i = -50 TO -25 -- a valid TO clause
FOR i = 50 TO 25 -- a NON-valid TO clause
FOR i = -25 TO -50 -- a NON-valid TO clause
```



Note: If the counter variable has a value before the FOR loop activation, this value will be replaced by the *first* value as soon as the program starts iterating the FOR loop. This allows you to use one and the same variables as counters for several loops which are not nested in one another.

The STEP Clause

The STEP clause is used to specify the number by which the counter value will be increased each time the loop is iterated. The STEP value is 1 by default, and you can omit the STEP clause, if it suits your purposes. However, you can change the STEP value by using the STEP keyword with an integer value following it. For example, you may make the counter increase by 2:

```
DEFINE i INT
FOR i = 1 to 10 STEP 2
DISPLAY "i=",i CLIPPED AT i,2
END FOR
```

In this case, the loop will be iterated 5 times, and the program will display 5 lines to the screen: "i=1", "i=3", "i=5", "i=7", "i=9".

You can use a negative integer as a STEP value, but in this case you will have to specify the *first* value as the one which is bigger than the *last* one, if you operate positive values for the counter. In this case, the FOR loop will be terminated as soon as the *first* value becomes equal to or less than the *last* one.

```
DEFINE i INT
FOR i = 10 to 1
STEP -2
DISPLAY "i=",i CLIPPED AT 11-i, 2
```



END FOR

In this case, the program will display 6 lines to the screen: "i=10", "i=8", "i=6", "i=4", "i=2". You can order the displayed values depending on the iteration number using expressions in the AT clause, but as during the first iteration i=10, to display them in the correct order you can use (11-i) expression as the lines indication.

If you specify negative integers as the limits for the counter and the negative STEP value, you should also change places of the maximum and minimum values:

```
FOR I = 25 TO 50    STEP -2    -- a NON-valid TO clause
FOR i = -50 TO -25 STEP -2    -- a NON-valid TO clause
FOR I = 50 TO 25    STEP -2    -- a valid TO clause
FOR i = -25 TO -50 STEP -2    -- a valid TO clause
```

If the initial relation between the *first* and the *last* values meets the conditions of the loop termination, 4GL will ignore the statements in the logical block and transfer the program control straight to the statement following the END FOR keywords.

The Declaration Clause

Like with the IF and the CASE statements, Q4GL gives you a possibility to specify spot variables that will be available only within the current FOR loop. If you want to do it, you have to add the declaration clause after the counter or after the STEP clause, if any. The declaration clause consists of one or more DEFINE statements. The rules and restrictions on the variables specified in this declaration clause are the same as those on the local variables. The names of the spot variables shouldn't coincide with the names of the variables of wider scope; otherwise, they may return unexpected values and result in incorrect program execution.

The spot variables can be referenced by a function program block, if it is invoked from the current FOR loop. Below is given an example of a declaration clause specified within a FOR loop:

```
MAIN
...
FOR j = 1 to 25
  DEFINE i INT -- declares a spot variable
  # calls a function to get the value
  # for the spot variable
  CALL get_i() RETURNING i
  DISPLAY "The current value is ", j at 3, 3
  LET j = j+i
  SLEEP 1
```



```
END FOR

END MAIN

FUNCTION get_i()

    DEFINE r_i INT

    PROMPT "What should the step be? " FOR r_i

    RETURN r_i -- returns the value to the spot variable i

END FUNCTION
```

In this example, the MAIN program block contains a FOR loop with a spot variable declaration. The value for the spot variable is returned by a function. The value of the variable *j* is changed each time the user inputs a new value for the variable *i* until the variable *j* gets the value which is equal to or less than 25.

The CONTINUE FOR Statement

Sometimes it may be useful to interrupt the current loop iteration and to start the next one. If you want the program to do this, use the CONTINUE FOR statement.

The CONTINUE FOR statement, when activated, forces the program to ignore all the statements between the CONTINUE FOR and END FOR keywords, to change the counter value, and to go back to the beginning of the loop, if the counter value doesn't force the loop termination. The CONTINUE FOR keywords should be located within a conditional statement (i.e. IF or CASE). If it is not in a conditional statement and placed immediately before the END FOR keywords, it will produce no effect; if between the END FOR keywords and such statement other statements appear, they will never be executed.

```
DEFINE i INT

FOR i = 1 to 10

    STEP 2

    IF i = 5 THEN

        CONTINUE FOR -- starts the loop iteration from the
                      -- very beginning, "i=5" will be omitted
        END IF

    DISPLAY "i=",i CLIPPED -- displays odd numbers from 1 to 9

END FOR
```

The EXIT FOR Statement

To terminate the loop iteration and to quit the FOR loop during one of the iterations, use the EXIT FOR statement. It is equal to other EXIT statements: it forces the loop termination ignoring all the statements specified between the EXIT FOR and the END FOR keywords. If the EXIT for statement is executed, the program execution resumes at the first statement following the END FOR keywords.



WHILE Loop

The WHILE loop directs the program to perform a set of actions while a condition specified after the WHILE keyword is TRUE. It is similar to the FOR loop, but is typically used when you do not know the number of iterations required. The general syntax of the WHILE loop is as follows:

```
WHILE expression
      [Declaration clause]
      ...
      ...
      ...
      }
      logical block
END WHILE
```

The WHILE loop begins with the WHILE and ends with the END WHILE keywords. At the first stage of the WHILE loop iteration the program evaluates the Boolean expression following the WHILE keyword. If it is TRUE, the program executes the statement block and evaluates the Boolean expression once more. When the expression is evaluated as FALSE, the WHILE loop is terminated, and the program execution resumes at the statement that follows the END WHILE keywords. If the Boolean expression gets the FALSE value during the loop iteration, the logical block is executed till the end, it does interrupt the loop execution. After that the conditional expression is evaluated and the loop is terminated.

Here is an example of a WHILE loop usage:

```
MAIN
      DEFINE a CHAR ( 5 ),
              i INTEGER
      LET i = 1
      LET a = "zzzzz"
      DISPLAY a
      WHILE a MATCHES "*z*" -- checks if the value contains any
                            --letters "z"
          LET a[i] = "a" -- replaces the [i] character by the
                            --letter "a"
          DISPLAY a
          LET i = i+1
      END WHILE
```



END MAIN

This part of source code checks the value of the variable *a* and replaces its characters by "a" letters until no "z" letters are left. After that the loop is terminated. Note that the value of a variable is not changed automatically by the WHILE statement as the FOR statement changes its counter. Thus you need to change the value of any WHILE condition explicitly, if you do not want to create an infinite loop.



Note: It is not necessary to assign an initial value for a counter if it is used in the FOR loop, but any of the variables that comprise the WHILE expression must have non-NULL value. Otherwise, the WHILE loop may become infinite or cause an error.

The CONTINUE WHILE statement is used to interrupt the logical block execution and to start it from the very beginning. The EXIT WHILE statement is used to terminate the WHILE loop iteration and to pass the program control to the first statement following the END WHILE keywords. These statements are similar to the CONTINUE FOR and EXIT FOR statements. The EXIT WHILE and CONTINUE WHILE statements should be placed within conditional statements, otherwise they may violate the result you are expecting.

You can nest several loops in one another, but such structure can become rather complicated. It is convenient to use the [GOTO](#) statement to exit the deeply nested loops:

```
LABEL retry:  
...  
WHILE i<200  
    FOR a = -5 to 5  
        ...  
        IF fin_val<10 THEN  
            GOTO: retry  
        ELSE  
            ...  
        END IF  
    END FOR  
END WHILE
```

The Declaration Clause

The WHILE statement can also contain a declaration clause. It is to be located between the *expression* and the logical block and specifies the spot variables available only for the statements within the current WHILE loop and within the functions invoked from this loop.



The following example demonstrates how a declaration clause can be used within the WHILE loop. The snippet of the source code is a modified version of the example code given in the *Declaration clause* section of the FOR statement:

```
WHILE j<=25

    DEFINE i int

    PROMPT "What should the step be? " FOR i

    DISPLAY "The previous value is ", j at 3, 3

    LET j = j+i

    DISPLAY "The current value is ", j at 3, 5

    SLEEP 1

    CLEAR SCREEN

END WHILE
```

The WHILE block contains a declaration clause specifying a variable used to store the user-defined value by which the value of the variable *j* is changed during each loop iteration while the variable *j* is less than or equal to 25.

Example

```
#####
# These are examples of conditional loops FOR and WHILE
#####

MAIN

DEFINE
    i,          -- the counter
    incr,       -- the increment
    iter,       -- the iteration number
    j    INTEGER

# To see the loops execution step by step remove the comment indicators from #
# the SLEEP statements below

DISPLAY "This is an example of a simple FOR loop:" AT 1,2
ATTRIBUTE (GREEN, BOLD)

LET iter = 0
```



```
FOR i = 1 TO 6 -- if no STEP clause is specified, the increment is 1 by
                -- default thus each next iteration i=i+1

        LET iter = iter + 1    -- here we specify the serial number of the
                               -- iteration to display
        LET j = i+1            -- this is how we define the variable which will
                               -- be used as the line indicator in the DISPLAY
                               -- statement below
        DISPLAY "iteration ",iter,", i = ",i AT j,2
#
        SLEEP 1
END FOR          -- 4GL goes back to the beginning of the FOR
                  -- statement and checks the i-counter. only if it
                  -- is greater than 6, it executes the statement
                  -- below END FOR

LET j = i+1
DISPLAY "after END FOR i = ",i AT j,2

LET iter = 0           -- here we set the number of iterations to zero
LET incr = 2           -- here we specify that increment should be 2
FOR i = 1 TO 6 STEP incr -- and here we use the increment in the STEP
                           -- clause thus each next iteration i = i+2

        LET iter = iter + 1
        LET j = i+1
        DISPLAY "iteration ",iter,", i = ",i AT j,25
#
        SLEEP 1
END FOR
LET j = i+1
DISPLAY "after END FOR i = ",i AT j,25

LET iter = 0
FOR i = 6 TO 1 STEP -2 -- the STEP clause can also have negative
                        -- increment, the increment can be either a literal
                        -- or a variable here each next iteration i=i+(-2)

        LET iter = iter + 1
        LET j = 8-i
        DISPLAY "iteration ",iter,", i = ",i AT j,50
#
        SLEEP 1
END FOR
LET j = 8-i
DISPLAY "after END FOR i = ",i AT j,50

SLEEP 10
DISPLAY "This is an example of the FOR loop with CONTINUE FOR and EXIT FOR:"
AT 9,2 ATTRIBUTE(GREEN, BOLD)
LET iter = 0

FOR i = 1 TO 6    -- here again i=i+1 in each new iteration
    LET iter = iter + 1

    # We include an IF statement into a loop to introduce CONTINUE FOR
    # It is advisable not to use these keywords outside a conditional
    # statement because it may lead to infinite loops.
```



```

        IF i >= 3 AND i <= 5 THEN
        LET j = i+9
        DISPLAY "CONTINUE FOR" AT j,2 ATTRIBUTE(YELLOW, BOLD)
#      SLEEP 1
        CONTINUE FOR -- if i has values from 3 to 5 the program control
                      -- is passed to the beginning of the loop and the
                      -- next iteration is started the statements
                      -- following the CONTINUE FOR statement are not
                      -- executed in this case
        END IF

        LET j = i+9
        DISPLAY "iteration ",iter,", i = ",i AT j,2
#      SLEEP 1
END FOR
LET j = i+9
DISPLAY "after END FOR i = ",i AT j,2

LET iter = 0

FOR i = 1 TO 6
    LET iter = iter + 1

    # We include an IF statement into a loop to introduce EXIT FOR
    # It is advisable not to use these keywords outside a conditional
    # statement. It may cause a part of your code to be constantly
    # ignored.
    IF i = 4 THEN
        LET j = i+9
        DISPLAY "EXIT FOR" AT j,25 ATTRIBUTE(YELLOW, BOLD)
#      SLEEP 1
        EXIT FOR -- if i = 4, the FOR loop will be terminated, all the
                  -- statements up to the END FOR will be skipped and the
                  -- program control will be passed to LET j = i+9 line
                  -- below
        END IF
    LET j = i+9
    DISPLAY "iteration ",iter,", i = ",i AT j,25
#      SLEEP 1
END FOR
LET j = i+10
DISPLAY "after END FOR i = ",i AT j,25

SLEEP 10
DISPLAY "This is the WHILE loop with CONTINUE WHILE and EXIT WHILE:" AT 17,2
ATTRIBUTE(GREEN, BOLD)

LET iter = 0
LET i = 0      -- we need to set i to zero as WHILE does not initiate its
               -- conditional variable automatically

WHILE i <= 6      -- the loop will be iterated while i value is 6 or less
    LET iter = iter + 1
    LET i = i + 1 -- we need this line because we need to change the value
                  -- of the condition manually. If it remains unchanged and

```



```
-- TRUE, the loop will become infinite and 4GL will never
-- exit it
LET j = i+17
DISPLAY "iteration ",iter,", i = ",i AT j,2
# SLEEP 1
END WHILE -- 4GL checks the value of i and if it is 6 or less it returns
-- the program control to the beginning of WHILE loop. If it is
-- greater than 6, the following LET statement executes

LET j = i+17
DISPLAY "after END WHILE i = ",i AT j,2

LET iter = 0
LET i = 0
WHILE i <= 6
    LET iter = iter + 1
    LET i = i + 1-- again we need to change the value of the WHILE condition

    # We include an IF statement into a loop to introduce CONTINUE WHILE
    # It is advisable not to use these keywords outside a conditional
    # statement
    IF i >= 2 AND i <= 4 THEN
        LET j = i+17
        DISPLAY "CONTINUE WHILE" AT j,25 ATTRIBUTE(YELLOW, BOLD)
    # SLEEP 1
    CONTINUE WHILE -- if i has values from 3 to 5 the program control is
                    -- passed to the beginning of the WHILE loop, the
                    -- rest of the statements up to the END WHILE are
                    -- skipped
    END IF
    LET j = i+17
    DISPLAY "iteration ",iter,", i = ",i AT j,25
# SLEEP 1
END WHILE

LET j = i+17
DISPLAY "after END WHILE i = ",i AT j,25

LET iter = 0
LET i = 0

WHILE i <= 6
    LET iter = iter + 1
    LET i = i + 1

    # We include an IF statement into a loop to introduce EXIT WHILE
    # It is advisable not to use these keywords outside a conditional
    # statement, this may cause a part of your code to be constantly
    # ignored.
    IF i = 5 THEN
        LET j = i+17
        DISPLAY "EXIT WHILE" AT j,50 ATTRIBUTE(YELLOW, BOLD)
    # SLEEP 1
    EXIT WHILE -- if i = 5, the WHILE loop will be terminated,
                -- all the statements up to the END WHILE will be skipped
```



```
-- the program control will be passed to LET j = i+17 line
END IF
LET j = i+17
DISPLAY "iteration ",iter,", i = ",i AT j,50
# SLEEP 1
END WHILE
LET j = i+18
DISPLAY "after END WHILE i = ",i AT j,50
SLEEP 10

# The rest of the code illustrates how variables can be declared within
# conditional loops
CALL fgl_window_open("win1",2,2,24,80,TRUE)
DISPLAY "A spot variable is declared inside the FOR loop" AT 10,15
ATTRIBUTES (GREEN, BOLD)
FOR i = 1 TO 3
    DEFINE pers_name CHAR (15) -- The variable pers_name is declared
                                -- within the FOR loop after the maximum
                                -- counter value. It is visible only to the
                                -- code inside this loop

    CALL names() RETURNING pers_name -- and in the function names() which
                                      -- we call from this loop

    DISPLAY i, ". ", pers_name CLIPPED AT i+4,4 ATTRIBUTE (BOLD, RED)

END FOR

# If you uncomment the next line, the program will not compile because the
# variable pers_name is out of its scope of reference now
# DISPLAY pers_name AT 9,9
SLEEP 2
CALL fgl_window_close("win1")

SLEEP 1

CALL fgl_window_open("win2",2,2,24,80,TRUE)
DISPLAY "Spot variables are declared within the WHILE loop" AT 5,15
ATTRIBUTES (GREEN, BOLD)
LET i = 0
WHILE i <= 5
    DEFINE num, num_g INT -- These variables are defined within the
                           -- WHILE loop, so they are visible only to the code
                           -- inside the loop
    LET num = 4
    CALL numbers() RETURNING num_g -- or to this function we call from
                                   -- within this loop
    IF num_g < 1 OR num_g > 5 OR num_g IS NULL THEN
        DISPLAY "You must enter a valid number!" AT 2,2
        ATTRIBUTES (RED,BOLD)
        SLEEP 2
        DISPLAY "                                     " AT 2,2
        CONTINUE WHILE
    END IF
```



```
IF i=5 AND num_g != num
THEN DISPLAY "No more attempts. The correct number is" AT 3,2
    ATTRIBUTES (RED, BOLD)
    DISPLAY num AT 3,42 ATTRIBUTES (BOLD)

ELSE IF num_g != num THEN
    DISPLAY "Wrong!" AT 3,2
    DISPLAY num_g AT 3,9 ATTRIBUTES (BOLD, RED)
    DISPLAY "is incorrect. Try again!" AT 3, 11
    SLEEP 1
        DISPLAY "
" AT 3,2
    ELSE IF num THEN
        DISPLAY "Right!" at 3,2
        DISPLAY num_g AT 3,9 ATTRIBUTES (BOLD, GREEN)
        DISPLAY "is correct!" AT 3,11
        EXIT WHILE
    END IF
END IF

END IF
LET i = i + 1
END WHILE
SLEEP 3
# If you uncomment the following line, there will be a compile error since
# the variable num is out of its scope of reference now
# DISPLAY "The correct number is ", num AT 4, 4

CALL fgl_window_close ("win2")

END MAIN

#####
# This function is called from within the FOR loop. The
# variable defined there is available here too
#####
FUNCTION names()
    DEFINE pers_name_f CHAR (15)
    PROMPT "Type the names of three people" FOR pers_name_f
    RETURN pers_name_f
END FUNCTION

FUNCTION numbers() -- This function is invoked from inside the WHILE loop.
-- The variable declared there is visible here too
    DEFINE num_guess INT
    PROMPT "Choose a number from 1 to 5" FOR num_guess
    RETURN num_guess -- The value of this variable will be passed back to
-- the WHILE loop from which the function was called
END FUNCTION
```



User Interaction

4GL supports a set of statements that allow the user to interact with the program and to influence its execution. These statements are called *user interaction statements*. In this chapter, we will discuss two of them - the PROMPT statement and the MENU statement.

The PROMPT Statement

The basic notion of the PROMPT statement has already been presented in "[Program Workflow Control](#)" chapter. There we discussed the role of PROMPT in the program workflow control. This statement is also used to receive values from a user which will be discussed in this chapter.

The simplest syntax of the statement as we already know is:

```
PROMPT "Character string" FOR variable
```

However, the PROMPT statement may include a number of clauses that determine the way it functions. The more complicated syntax of the PROMPT statement looks as follows:

```
PROMPT "Character string" [ATTRIBUTE (attribute,...)] FOR [CHAR]
      receiving_variable [HELP number] [ATTRIBUTE (attribute,...)]
      [ON KEY (key1, key2...) clause]
      [BEFORE PROMPT clause]
      [AFTER PROMPT clause]
      [END PROMPT]
```

The clauses taken in brackets are optional and may be used in any combinations. However, their order should be kept.

The END PROMPT keywords are needed if the ON KEY, BEFORE PROMPT, or AFTER PROMPT blocks are present in the PROMPT statement in any combination. The END PROMPT keywords are used to separate the statements of these blocks from the other statements of the program and to indicate the end of the PROMPT statement.

The Receiving Variable

The variable that gets the value after the user enters it and presses the RETURN key, is called a *receiving variable*. The receiving variable can be any variable of a simple data type. It can be also of a structured data type which we will look upon in the following chapters. A run-time error occurs, if the value entered and the receiving variable are of incompatible data types. If the receiving variable is of a character data type, its value may contain blank spaces only.



The FOR Clause

The receiving variable is identified by means of the FOR clause, which is the only obligatory clause of the RETURN statement. The CHAR keyword may optionally follow the FOR keyword. It will allow the user to enter a single-character value which will be immediately assigned to the receiving variable and the PROMPT statement will be terminated without the RETURN key being pressed.

The following source code sample checks the values, sent to the receiving variable and chooses what statement to execute on the basis of the obtained result:

```
MAIN

    DEFINE yes_no CHAR --the variable to be checked

    LABEL restart: -- the label used to restart the PROMPT statement
                  -- if the entered value is invalid
    PROMPT "Do you want to exit the program? (y/n) " FOR CHAR yes_no
    CASE           -- checks the possible values and chooses the
                  -- statements to execute
        WHEN yes_no MATCHES "[Yy]" EXIT PROGRAM
        WHEN yes_no MATCHES "[Nn]" GOTO follow
        OTHERWISE DISPLAY "You pressed the wrong key" AT 1,1
        SLEEP 2
        GOTO restart
    END CASE

    LABEL follow:
    DISPLAY "The program continues its execution" AT 2,2
    SLEEP 2
END MAIN
```

The user won't have to press the RETURN key in order to pass the value to the receiving variable.

The HELP Clause

The HELP clause is used to specify a literal integer ranging from -32,767 to 32,767 that corresponds to the number of the HELP message for the current PROMPT statement. If the user wants to read the HELP message, he or she should press the HELP key (CONTROL-W by default), and the message will be displayed on the screen.



The Help Key

You can change the default help key (which is CONTROL-W) using the OPTIONS statement. The syntax of the OPTIONS statement is as follows:

```
OPTIONS HELP KEY "key"
```

Here "key" is the keyword or the name of the key assigned. This can be F1 through F256 function keys, CONTROL-char combinations, etc. The keys or key combinations which can be used as the help key are the same which are used in the [ON KEY block](#) of the PROMPT statement which is discussed later in this chapter.

The Help File

If you want the program to refer to help messages, you have to create a message file containing a list of numbered HELP messages and compile it by means of Informix Tools so that it has extension .erm. The HELP clause can refer only to .erm files. Message files (e.g., help_file.msg) that haven't been compiled and converted into .erm files won't be recognized by the HELP statement.

A message file content may look as follows:

```
.101
This is the first help message

.102
This is the second help message

.103
This is the third help message
```

The numbers before the messages are those referenced by the HELP clauses with the initial dot omitted.

You should specify the name of the HELP file in the HELP FILE clause of the OPTIONS statement before you use the HELP clause:

```
MAIN
DEFINE...
OPTIONS HELP FILE "filename.erm"
...
...
PROMPT "Message" FOR variable HELP 103
```

The OPTIONS statement with the help file specification must appear earlier within the source file than the PROMPT with the HELP clause.

A runtime error will occur, if you refer to a HELP message number that is not specified in the HELP file, or if the HELP file cannot be opened by 4GL due to any reasons.



The ATTRIBUTE Clause

The ATTRIBUTE clause defines the display attributes for the PROMPT message which temporary override the attributes specified in OPTIONS and OPEN WINDOW statements. The syntax of the ATTRIBUTE clause is:

```
ATTRIBUTE (attribute1 [, attribute2,...])
```

As you can see from the PROMPT statement syntax, there may be two ATTRIBUTE clauses. The first one is used to specify the display attributes, applied to the prompt message displayed. The default value of this attribute is NORMAL.

The second attribute clause is used to specify the display attributes, applied to the prompt input field where the value is entered by the user. The default value of this attribute is also NORMAL.

```
PROMPT "This is prompt message" ATTRIBUTE (YELLOW)
FOR variable ATTRIBUTE (GREEN, REVERSE)
```

The second ATTRIBUTE clause of the PROMPT statement can include the CENTURY attribute, if the receiving variable is of DATE or DATETIME data type. The CENTURY attribute is used to define how the program should expand one- and two- digit year parts of the DATE and DATETIME values.

The syntax of this attribute is as follows:

```
CENTURY = "Symbol"
```

where *symbol* stands for one of the four symbols that specify the algorithm of expansion. These symbols and their meanings are represented in the table below:

Symbol	Meaning	Algorithm of Abbreviated Years Expansion
C or c	Closest	Use the year that is closest to the current date, regardless whether it will be past, current, or future year
F or f	Future	The nearest year is the future is used
P or p	Past	The nearest year in past is used
R or r	Current	The current year is used

Here is an example of the CENTURY attribute application:

```
PROMPT "You were born in: " FOR birth ATTRIBUTE (CENTURY = "p")
```

Most people who will enter values to this PROMPT line now, at the very beginning of the XXI century, were born in the XX century. So, if they enter such values as "54", "87", or "93", the program will automatically add "19" preceding the entered value.

The ON KEY Clause

The ON KEY block is used to specify one or several statements that will be executed when the user presses one of the keys, given in parentheses following the ON KEY keywords, during the execution of the PROMPT statement. When the ON KEY block is executed, the program control is passed to the statements that follow the END PROMPT keywords, and the statements between the ON KEY and the END PROMPT keywords are ignored. In this case, the value of the receiving variable remains undetermined, if nothing has been entered into the prompt field. If some value has been entered into the prompt field, but an ON KEY key has been pressed before the RETURN key, the value contained by the prompt field will be assigned to the receiving



variable. Thus the user can press only one key specified in the ON KEY block during the PROMPT statement, even if you specify more than one ON KEY clause, because after that the PROMPT is terminated.

The general syntax of the ON KEY block is:

```
ON KEY (keyname) statement
```

The *keyname* may have the following values (which are case insensitive):

Key Name	The Key	Special features and restrictions
ACCEPT	Escape	
DELETE	Delete	
DOWN	DOWN arrow	
UP	UP arrow	
LEFT	LEFT arrow	
RIGHT	RIGHT arrow	
ESC or ESCAPE	Escape	To set a new action for the Escape key, you should first define a new key as the ACCEPT by means of the OPTIONS statement
F1 through F256		
HELP	CONTROL-w	
NEXT or NEXTPAGE	F3 , F4	
INSERT	F2	
INTERRUPT	CONTROL-c	If you want to use these keys in the ON KEY block, you should activate DEFER QUIT or DEFER INTERRUPT in order to prevent the program termination.
QUIT	CONTROL-\	
TAB	TAB, SHIFT-TAB	
CONTROL-<i>char</i>	except A,D,H,I,J,K,L,M,R, or X	

You probably won't be able to use some of these keys, if they have special meaning in your system.

Here is a very simple example of the PROMPT statement with the ON KEY clause in a 4GL program:

```
MAIN

DEFINE y_word CHAR (20)

DISPLAY "Press F5 to exit" at 4,2

PROMPT "Enter a word " FOR y_word

    ON KEY (F5)
        DISPLAY "You quit the prompt"
        SLEEP 2
        EXIT PROGRAM

END PROMPT
```



```
DISPLAY "You entered the word ", y_word at 4,2
SLEEP 2
END MAIN
```

If the user presses the F5 key the value entered by the user to a variable *y_word*, the warning message is displayed and then the program is terminated.

The BEFORE PROMPT Clause

The BEFORE PROMPT control block specifies what actions the program should take before the user is allowed to enter data into the PROMPT field. You can use this clause to display some information for the user, for example, the notion of the restrictions on the data to be entered, what keys can be pressed to quit the PROMPT statement and to view the help file, etc.

The PROMPT statement can contain only one BEFORE PROMPT clause.

In the following example, the BEFORE PROMPT statement is used to tell the user what data they are to enter to the PROMPT field:

```
DEFINE a INT
PROMPT "ENTER a " FOR A
ON KEY (F5) DISPLAY "You've terminated the PROMPT
statement " at 2,2
SLEEP 2
BEFORE PROMPT
DISPLAY "a is an integer variable" at 2,2
END PROMPT
```

The EXIT PROMPT Keywords

The EXIT PROMPT statement is used to terminate the PROMPT statement. When this happens depends on the position of the EXIT PROMPT statement, e.g. if it is located in the BEFORE PROMPT statement, the PROMPT terminates before the user is able to enter values. In the ON KEY clause it causes the statement to terminate, if the user presses a certain key. The EXIT PROMPT statement can be used within the BEFORE PROMPT which checks the current value of the variable and decides whether it should be changed:

```
PROMPT "Enter b: " FOR b
BEFORE PROMPT
IF b<15 THEN -- checks the actual value of the variable b
EXIT PROMPT
```



```
END PROMPT
```

In this case the PROMPT statement terminates before the user is able to enter any value, if the value of b is less than 15.

The AFTER PROMPT Clause

The AFTER PROMPT clause is used to specify the actions that are to be taken after the user presses the RETURN key and the entered value is assigned to the variable.

If you have added the CHAR keyword to the PROMPT statement, the AFTER PROMPT block is executed after the user presses any key. The program ignores the AFTER PROMPT statement, if the user presses any key specified in the ON KEY block or if the EXIT PROMPT statement is executed.

The PROMPT statement can contain only one AFTER PROMPT block.

In the following example, the AFTER PROMPT statement displays a message for the user:

```
DEFINE a INT  
  
PROMPT "ENTER a " FOR A  
  
ON KEY (F5) DISPLAY "You've terminated the PROMPT  
statement " at 2,2  
  
SLEEP 2  
  
BEFORE PROMPT  
  
DISPLAY "Press F5 to quit" at 2,2  
  
AFTER PROMPT  
  
DISPLAY "Thank you for entering the data" at 2,2  
  
END PROMPT
```

Opening a Prompt in a Separate Window

Sometimes it is convenient that the prompt line appears not in the current screen or window, but in a special dialog box. You can use the *fgl_winprompt()* function to display a dialog box with a prompt field.

The syntax of the function invocation is:

```
fgl_winprompt (x, y, "text", "default", length, type)
```

- The *x* and *y* arguments stand for integer values that specify the column and line position of the dialog box.
- *Text* stands for the character value containing the message that will appear within the message box.
- *The default* stands for the character string specifying the default text displayed to the input field. This value must be compatible with the *type* argument.



- The *length* argument is an integer value specifying the maximum length of the input field.
- The *type* argument is an integer value specifying the data type code. The argument can get the following values: 0 = CHAR, 1 = SMALLINT, 2 = INTEGER, 7 = DATE, 255 = invisible.

The function returns the value entered into the prompt field. As this function returns a value, you should either use it as the right hand argument of the LET statement, or invoke it with the CALL statement which contains the RETURNING clause.

The following example demonstrates how a dialog box can be created to assign a value to the variable *nam*:

```
MAIN

DEFINE nam CHAR(10)

LET nam = fgl_winprompt(2,2, "Enter your name", "", 10, 0)

DISPLAY nam at 2,2

SLEEP 2

END MAIN
```

It is advisable, that the *type* argument and the data type of the variable which receives the value returned by the function were of compatible data types. Otherwise the value entered into the prompt field will not be assigned to the variable receiving the returned value, if it cannot be converted to its data type.

Preventing an Application From Closing During Input

Sometimes it is advisable to prevent the user from closing the application until the input is not finished. We can use the OPTIONS ON CLOSE APPLICATION statement to prevent the application from closing. We have already discussed some features of this statement and will now discuss features which are only applicable, if the user presses the Close button (x) during the input. These features are:

```
OPTIONS ON CLOSE APPLICATION CONTINUE
OPTIONS ON CLOSE APPLICATION STOP
OPTIONS ON CLOSE APPLICATION KEY key_name
```

- The CONTINUE statement does not prevent the application from closing. However, if it is specified and the Close button (x) is pressed during user input, the statement used for input (e.g. the PROMPT statement) is considered to be executed with no values entered.
- The STOP option enables the Close (x) key and a user is able to close the application using it.
- The KEY statement following the OPTIONS ON CLOSE APPLICATION statement is used to specify the keypress that will be sent to the program, if the user presses the Close button (x). In other words, when you specify the KEY *key_name* option, the program acts as if the user has pressed the specified key instead of the close button.



You do not need any special code arrangements, if you specify CONTINUE. However, if you specify the key which should be pressed instead of the Close button (x), you need to assign some action to that key. It can be done in an ON KEY clause of the PROMPT statement active at the time the Close button is pressed. The same can be done for any other statements used for input which will be discussed later in this manual. E.g.:

```
OPTIONS ON CLOSE APPLICATION KEY F1
...
LABEL: no_close
PROMPT "Enter something " FOR i
ON KEY (F1)
    GOTO no_close
END PROMPT
```

If the user presses the Close button (x) in the example above, he will be returned to the PROMPT statement and the application will not be terminated.

In the following example the Close button (x) is redirected to the default help button and pressing it will open the help instead of closing the application:

```
OPTIONS ON CLOSE APPLICATION KEY CONTROL-W,
    HELP FILE "my_help erm"
...
PROMPT "Enter something " FOR i HELP 100
```

Keep in mind that the key to which you redirect the Close button (x) should be specified in the source code either in the corresponding ON KEY clause or in any other way (i.e. the default help key has no effect, unless the input statement during which the user tries to close the application has no HELP clause).

The MENU Statement

The MENU statement creates and displays a ring menu to the 4GL screen or window. The displayed menu takes two lines on the screen. The first line is reserved for the menu title and a set of menu options. Each option, when selected and activated, makes the program execute a specified set of statements. The second line is reserved for the options descriptions that appear when the cursor highlights one of the options. The default position of the menu is the first and the second lines.

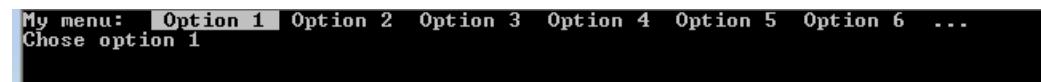
The menu can be placed above a screen form or below it, but you can't locate it within a form.

You can nest one MENU statement into another one. A nested MENU block may be located within a statement block of another MENU or within a function that is called by one of the MENU control block statements.



The user can chose the menu options by pressing arrow keys and spacebar. The LEFT and the UP arrow keys move the cursor from right to left; the spacebar and the RIGHT and DOWN arrow keys make the cursor move from left to right. The RETURN key is used to activate the selected menu option.

Sometimes the list of the menu options is too long and cannot fit the screen size. If so, a *multi-page* menu is automatically created by 4GL: in character mode, a (...) sign appears to the right of the options that fit into the screen.



The screen in GUI mode can include more characters, and if there is still the need for a ring menu, the arrow buttons at the each side of the menu allow the user to get to the next portion of the menu. Therefore, more options will fit the screen. A ring menu in a GUI mode looks as follows:



When the cursor is positioned to the first option that didn't fit the screen, the option list "pages". Below are the pictures showing the options paging in character and in GUI mode:



When a cursor moves beyond the last option, it comes back to the first one; that is why such menu is called a *ring menu*.

The MENU block must start with the MENU keyword and end with the END MENU keywords. The general syntax of the MENU statement is as follows:

```
MENU "title" [BEFORE MENU block] Control block END MENU
```

The menu *title* is a quoted string indicating a display label for the ring menu. If you don't want to display the menu title, make the character string empty (i.e. " "). The character string that specifies the menu title may be replaced by a variable of CHAR or VARCHAR data type.

The menu title is displayed only if the application is run in the character mode. When it is run in the GUI mode, no menu title is displayed:



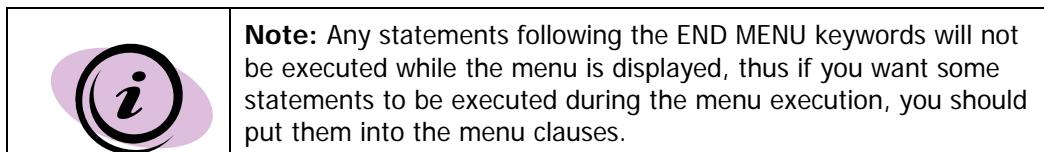
My menu: **Option 1** - The menu named *My menu* is displayed in character mode.

Option 1 - The same menu when the application is run in Phoenix

The statements between the MENU and the END MENU keywords are called *menu control block*. The control block consists of an optional BEFORE MENU block and at least one COMMAND block with one or more executable statements.

When the MENU statement is activated, it waits until the user chooses one of the options, and executes the statements located in the statement block of the selected option. When all the statements of an option statement block are executed, the program redisplays the menu, if no other actions are specified.

The MENU statement is interrupted when the user presses the INTERRUPT key or the program executes the EXIT MENU statement.



The COMMAND Block

The COMMAND block consists of a menu option declaration and a list of statements that are to be executed when the user selects this option. The general structure of a typical COMMAND BLOCK is:

```
COMMAND [KEY(key)] option_name      [option_description]  [HELP number]
...          } Statement block
...
[END MENU]
```

Option Name and Description

The *option name* is used to indicate the option name that will be displayed to the list of options and which can be highlighted with the cursor.

The *option description* is an optional character string that appears under the menu options and describes the option which is currently highlighted with the cursor. The option description can take only one line of the 4GL screen or window.

Both option name and description may be specified by a quoted string, a variable, or a 4GL expression. Here is an example of a simple menu:



```
MAIN
DEFINE op_name CHAR (10),
      op_descr CHAR (20)

LET op_name = "Option 1"

LET op_descr = "Chose option 1"

MENU "My menu" -- the menu name

COMMAND op_name op_descr -- the option name and description represented
                           -- by variables

DISPLAY "You have chosen the Option 1"

SLEEP 2

COMMAND "Option 2" "Chose option 2" -- the option name and description
                           -- are represented by character strings

DISPLAY "You have chosen the Option 2"

SLEEP 2

END MENU

END MAIN
```

From the menu in the example above you can exit only by pressing the Interrupt key or Quit key (CONTROL-C or CONTROL-\).

The option name and the option description can be represented by values of one or several variables and/or character strings and include operators. If you want them to be specified this way, you have to enclose them in parentheses (e.g., COMMAND ("User" || var2), "Chose this option in order to...")

The option name and option description can be separated by a comma, when declared. If you separate them with a comma, you have to specify both option name and description. If you use no separator, you can specify only the option name. It's impossible to specify an option description without an option name: if only one value is present in the command block, it is treated as an option name.

The KEY Clause

The *KEY clause* is optional and is used to specify one or more activation keys that will allow the user to activate the menu option without using the cursor. The syntax of the KEY clause is as follows:

```
KEY (key [,key...] )
```

The *key* must be unique within one MENU block. You can specify an activation key using both upper- and lower-case letters (the clause is not case sensitive), quoted symbols (such as !, @,#), and any of the key KEYWORDS (see the list given in the [ON KEY block](#) part of the PROMPT statement description). You can specify up to 4 activation keys in one KEY clause, they should be separated by commas.



Here is an example of the KEY clause usage:

```
COMMAND KEY (n, L) op_name op_descr  
COMMAND KEY ("@") "Option 2" "Chose option 2"
```

If you replace the corresponding lines of the previous example by these lines, you will be able to activate the *Option 1* by pressing the key *n* or *L*, and to activate the *Option 2* by pressing SHIFT-2(which indicates the "@" symbol).

If you specify no activation key, a default one will be set by the program, and it will be the first letter of the option name. The default option keys may have no effect, if several options have names that begin with the same letter.

The HELP Clause

The *HELP* block is similar to the one of the [PROMPT](#) statement and is used to specify the number of the help message that will be displayed when the user presses the help key (CONTROL-w). Each COMMAND block can have its own help message. The HELP clause must be specified after the option name or after the option description identifier, if any. To view the help message of a specific help option, highlight it with the cursor and then press the help key (CONTROL-W).

Invisible Options

Sometimes it is useful to make some options invisible. If you want to do it, you shouldn't specify the option name and description, but you have to specify an activation key for it:

```
MENU "My menu"  
  
COMMAND KEY ("!")  
  
DISPLAY "You have chosen the Invisible option"  
  
SLEEP 2  
  
COMMAND "Main" "Main option" -- option name and  
-- description are presented as  
-- character strings  
  
DISPLAY "You have chosen the Main option"  
  
SLEEP 2  
  
END MENU
```

If you run this menu, you will see only one option, named "Main". This option can be activated by pressing the RETURN key or the letter *m*. However, an invisible option is present in this menu, and this option can be called by pressing SHIFT-1 (which corresponds to the exclamation mark symbol (!)).



MENU Execution Control Statements

You can control the execution of the menu statement in several ways. For example, you can include the option which will exit the menu or hide some menu options. The ways of menu manipulation are described below.

The NEXT OPTION Keywords

The NEXT OPTION keywords are used to indicate which one of the menu options will become highlighted after the current option is executed. If you don't specify the next option, the current one will be the option that has just been executed.

You can't use the NEXT OPTION statement to activate a menu option; you can just make the cursor highlight the menu option:

```
MAIN

MENU "Structure"

COMMAND "Large units"

DISPLAY "Large units"

SLEEP 2

NEXT OPTION "Small units"

COMMAND "Mid-size units"

DISPLAY "Mid-size units"

SLEEP 2

COMMAND "Small units"

DISPLAY "Small units"

SLEEP 2

END MENU
END MAIN
```

If the user activates the option "Large units", a message appears for two seconds, and when the menu is redisplayed, the cursor moves automatically to the option "Small units", but does not activate it.

The CONTINUE MENU Statement

The CONTINUE MENU keywords are similar to other CONTINUE keywords used by 4GL. When the program encounters the CONTINUE MENU keywords, it ignores all the statements between them and the END MENU keywords, and redisplays the menu.



The EXIT MENU Statement

The EXIT MENU statement forces the program to terminate the menu and to ignore all the statements that are located between the EXIT MENU and the END MENU keywords:

```
MENU "MAIN"  
...  
...  
COMMAND "Exit" "Leave the Main menu"  
EXIT MENU  
END MENU
```

After the EXIT MENU keywords are executed, 4GL exits the menu and proceeds to the first statement following the END MENU keywords. The menu is no longer displayed. You should add at least one EXIT MENU statement to the menu control block in order to give the user a capability to quit the menu. If you do not include the EXIT MENU keywords in your menu, the user will be able to exit the menu only together with terminating the program by using the Quit or Interrupt key.

The HIDE OPTION and SHOW OPTION Keywords

The HIDE OPTION keywords are used to identify which menu options will be invisible and unavailable for the user. The HIDE OPTION keywords can be applied only to the options that have names. The SHOW OPTION keywords are used to show the options previously hidden by the HIDE OPTION keywords. The HIDE OPTION and SHOW OPTION syntax is very simple:

```
HIDE OPTION "Option name"  
SHOW OPTION "Option name"
```

You can use variables as option names indicators.

If you want to hide or to show all the menu options, you can put the ALL keyword after the HIDE OPTION or SHOW OPTION keywords.

You should not confuse the *invisible* options and the *hidden* ones. The invisible options don't have names and they cannot be displayed by means of the SHOW OPTION keywords, but they can be activated by means of activation keys. The hidden options cannot be activated even with the activation keys unless they are redisplayed to the menu by means of the SHOW OPTION statement.

The BEFORE MENU block

The program executes the statements that contained in the BEFORE MENU block before the menu is displayed:

```
MENU "My menu"  
BEFORE MENU  
DISPLAY "This is my first menu"  
SLEEP 1
```



COMMAND . . .

The string "This is my first menu" will appear in the screen for one second before the menu is displayed.

The BEFORE MENU block is typically used in order to:

- Specify the variables that will represent the menu name, option names, and option descriptions
- Check whether the user has all the required permissions
- Hide or show some menu options

If the EXIT MENU keywords are executed in the BEFORE MENU block, the menu won't be displayed at all. In the following example, the BEFORE MENU clause checks the name of the user, and displays different number of menu options depending on the result: the whole menu will be displayed if the user name is *admin*, the part of the menu will be displayed if the user name is *guest*, and the menu won't be displayed, if the user is somebody else:

```
MAIN

DEFINE u_name CHAR( 20 )

MENU "Main"

BEFORE MENU          -- checks the user access privileges
                      -- before the menu is displayed

PROMPT "Enter the user name: " FOR u_name

IF u_name NOT MATCHES "admin"

THEN

    IF u_name MATCHES "guest"

        THEN HIDE OPTION "Main"

        ELSE EXIT MENU

    END IF

END IF

COMMAND "Main" "The main menu option"
DISPLAY "You've chosen the MAIN option"
SLEEP 2

COMMAND "Time" "Press to know the current time"
DISPLAY CURRENT
SLEEP 2

COMMAND "Exit" "Exit menu"
EXIT MENU
```



```
END MENU
```

```
END MAIN
```

Setting Icons for the Menu Options

A standard menu consists of a number of keys with the option names written on them. It is possible to have menu buttons with icons instead of the buttons with labels. For this purposes the fgl_setkeylabel() function is used which is typically included into the before menu clause. This function has the following syntax:

```
fgl_setkeylabel("key_name", "label", "icon", position)
```

The key name argument refers the key specified in one of the KEY clauses of your menu which label and/or icon you want to change. The label argument specifies the option name which will be displayed on the menu button. The icon argument refers to an image file which is to be used as the icon. The position argument specifies the position of the menu option in the menu line.

This is possible only if your menu commands include the KEY clause which specifies the activation key for which this function can set an icon. This function can also change the name of the menu option without assigning the icon to it, if you leave the icon argument empty, or it can assign only icon and omit the label, if you leave the label argument empty.

In the example below the menu options will be displayed with labels and icons. Note that the COMMAND clauses do not include either option name or description. If you include them, you will have a duplicated menu with icons above the buttons with labels, both sets will be fully functional but will not match in length and will take up more of the screen space.

```
MENU "Icons"
  BEFORE MENU
    CALL fgl_setkeylabel("F1", "Add", "add.ico", 1)
    CALL fgl_setkeylabel("F2", "Delete", "delete.ico", 2)
    CALL fgl_setkeylabel("F3", "Exit", "exit.ico", 3)

    COMMAND KEY (F1)
    ...
    COMMAND KEY (F2)
    ...
    COMMAND KEY (F3)
    ...
  END MENU
```

This may look as follows in a GUI client:





The Position of the Menu and Prompt Lines on the Screen

By default, the prompt line and the menu options list are displayed to the first row of the 4GL screen. However, you can change their positions. To do it, you should use the PROMPT LINE and the MENU LINE keywords followed by a value which indicates the row where the prompt line will be displayed. This value may be presented by:

- A literal positive integer
- A variable or expression that returns a literal positive integer
- FIRST keyword, that indicates the first line of the screen
- FIRST + *integer* (a literal positive integer or a variable that returns it)
- LAST keyword, that indicates the last line of the screen
- LAST - *integer* (a literal positive integer or a variable that returns it)

The prompt and menu line specifications may be specified in:

- An attribute of the OPEN WINDOW statement in the form of the PROMPT LINE or MENU LINE:

```
...
OPEN WINDOW my_win AT 3,6 WITH 10 ROWS, 60 COLUMNS
ATTRIBUTE (BORDER, PROMPT LINE 5) --the prompt line is
-- specified here
PROMPT "Enter your name " FOR y_name
...
```

As the result of these lines execution, the prompt line will be displayed to the 5th row of a newly opened window.

- A part of the OPTIONS statement:

```
MAIN
DEFINE y_name CHAR (20)
OPTIONS PROMPT LINE 4, MENU LINE Last-2
PROMPT "Enter the Menu name " FOR y_name
MENU y_name
COMMAND "Time" "Shows the time"
DISPLAY Current
sleep 2
COMMAND "Date" "Shows the date"
DISPLAY TODAY
sleep 2
COMMAND "Exit" "Exit the program"
```



EXIT PROGRAM

END MENU

END MAIN

In this program, the prompt line will be displayed to the fourth line of the 4GL screen, and the menu will be displayed at the bottom of the screen.



Note: You shouldn't specify the menu line as *last*, because the menu needs two lines, so the program won't be able to display the description line.

Example

The following example illustrates the functionality of the MENU statement with nested menus, invisible and hidden options, etc. It also illustrates different ways to use the PROMPT statement.

```
#####
# This example illustrates the basics of the menu management and an enhanced
# prompt statement
#####

MAIN
  DEFINE
    answer          CHAR,
    answer_long     VARCHAR(15),
    winpr_answer    STRING,           -- This variable will store the value
                                    -- entered by the user into the prompt
                                    -- field in the dialog box which is invoked
                                    -- by the fgl_winprompt() function

    op_name1,
    op_name2        VARCHAR (20),   -- We will use these variables to store
                                    -- character strings. The variables will
                                    -- be concatenated to make an option
                                    -- name in the menu
    but_pressed     STRING          -- This variable will store the value
                                    -- returned by the fgl_winquestion()
                                    -- function to be evaluated by
                                    -- conditional statements

  LET op_name1 = "Form"   -- You must initialize these variables before
  LET op_name2 = "Call"  -- the MENU block begins

OPTIONS             -- if you have more than one clause in the
                    -- OPTIONS statement, they should be separated
                    -- by commas
```



```
MENU LINE FIRST + 1,      -- here we specify the menu lines as the second
                           -- and the third lines of 4GL screen
HELP FILE "my_help erm", -- here we specify the help file to be used
                           -- (pay attention to .erm extension)

HELP KEY "CONTROL-W", -- here we specify the key to invoke help (in our
                           -- case we specify the default key)
PROMPT LINE FIRST+4      -- we move PROMPT down several lines, because by
                           -- default it occupies the first line and we need
                           -- the first line for other purposes

DISPLAY "Here is an example of a ring menu:" AT 1,2 ATTRIBUTE(GREEN, BOLD)
        -- as we display this at the first line we
        -- needed to move the menu lines down a bit

MENU "First MENU"         -- this is the menu name.
                           -- It is visible only in the character mode
                           -- and not displayed in the GUI mode

BEFORE MENU               -- the BEFORE MENU clause is used here to display
                           -- an introductory message

DISPLAY "Before displaying the menu, 4GL displays this message, because"
AT 11,4
DISPLAY "it executes the statement block that follows the optional"
AT 12,4
DISPLAY "BEFORE MENU clause. Press any key."
AT 13,4
CALL fgl_getkey()

# then we clean the message from the screen
DISPLAY "
AT 11,4
DISPLAY "
AT 12,4
DISPLAY "
AT 13,4

# This is the first command
COMMAND "Prompt" "PROMPT and CONTINUE MENU statements demonstration"

#The PROMPT message below is executed, if you activate the "Prompt" menu
# option
LABEL lab1:
LET answer = ""
OPTIONS PROMPT LINE FIRST + 10 -- here we move the PROMPT message to
                           -- the eleventh line

PROMPT "                      Do you want to run the long report? (y/n)"
FOR CHAR answer ATTRIBUTE(CYAN, BOLD) -- this is an example where only
                           -- one character can be entered
                           -- into the prompt field
```



```
IF answer MATCHES "[Yy]"      -- 4GL executes the code depending on
                                -- the single character value entered
                                -- for the PROMPT
THEN CALL FGL_DRAWBOX(5,40,9,24,0)
DISPLAY "Running report. Please, wait..." AT 11,28 SLEEP 2
CLEAR SCREEN
DISPLAY "Here is the example of a simple ring menu:" AT 1,2
ATTRIBUTE(GREEN, BOLD)

CONTINUE MENU                  -- When 4GL executes these keywords,
                                -- the menu is redisplayed.
ELSE IF answer NOT MATCHES "[YyNn]" OR answer IS NULL THEN
DISPLAY "Please enter y or n" AT 11,28 SLEEP 2
DISPLAY "                         " AT 11,28
GOTO lab1
ELSE CONTINUE MENU
END IF
END IF

COMMAND "Submenu" "The option invokes nested menu in the 4GL Screen"
# The nested menu is just another MENU statement specified in one of the
# COMMAND clauses of the previous menu

OPTIONS MENU LINE FIRST + 5 -- we need to move the nested menu down
                            -- in order that it did not overlap with
                            -- the first level menu

MENU "Submenu" -- the second level menu is opened
                -- in the 4GL screen
COMMAND "Add" "Add a row to the table."
DISPLAY "Adding a row to the table." AT 10,2
ATTRIBUTE (GREEN, BOLD)
COMMAND "Change" "Update a row in the table."
DISPLAY "Updating a row in the table." AT 10,2
ATTRIBUTE (YELLOW, BOLD)
COMMAND "Delete" "Delete a row from the table."
DISPLAY "Deleteing a row from the table." AT 10,2
ATTRIBUTE (RED, BOLD)
COMMAND "Exit" "Return to the previous menu."

OPTIONS MENU LINE FIRST + 1 -- we return the menu onto the second
                            -- line of the 4GL screen, because we
                            -- want to see the first level menu
                            -- there when it is redisplayed

EXIT MENU                    -- 4GL will exit the submenu, then it will
                            -- continue executing the "Submenu"
                            -- command of the first level menu. When it
                            -- is executed, it will redisplay the first
                            -- level menu.

END MENU
CLEAR SCREEN
DISPLAY "Here is the example of simple ring menu:" AT 1,2
ATTRIBUTE(GREEN, BOLD)
```



```
COMMAND "Windowed Menu"
        "This option invokes a second level nested menu in a 4GL window."

# This nested menu is called within a window, so we do not need to change the
# menu position. The rest of the menu options are the same as in the
# "_submenu".

OPEN WINDOW w_win1 AT 5,20 WITH 16 ROWS,60 COLUMNS ATTRIBUTE(BORDER)

MENU "Window MENU"
    # We set the icons for the menu options using the built-in function
    # in the BEFORE MENU clause
    BEFORE MENU
        CALL fgl_setkeylabel("F9", "Add", "new01.ico", 1)
        CALL fgl_setkeylabel("F10", "Update", "edit01.ico", 1)
        CALL fgl_setkeylabel("F11", "Delete", "delete01.ico", 1)
        CALL fgl_setkeylabel("F12", "Exit", "exit01.ico", 1)

    # The options for which the labels are set have no option name and
    # no description, but they must have the KEY clause

    COMMAND KEY (F9)
        DISPLAY "Adding a row to the table."      " AT 8,2
        ATTRIBUTE (GREEN, BOLD)
    COMMAND KEY (F10)
        DISPLAY "Updating a row in the table."    " AT 8,2
        ATTRIBUTE (YELLOW, BOLD)
    COMMAND KEY (F11)
        DISPLAY "Deleting a row from the table." " AT 8,2
        ATTRIBUTE (RED, BOLD)
    COMMAND KEY (F12)
        EXIT MENU
    END MENU           CLOSE WINDOW w_win1

# The next option name is made up of two VARCHAR variables we defined and
# initialized before. They are concatenated by the double pipe operator. Note
# that they must be enclosed in parentheses and separated from the option
# description by a comma

COMMAND (op_name1||op_name2), "This option opens a window with form."
    OPEN WINDOW w_win2 AT 5,30 WITH FORM "form_1"
        ATTRIBUTE(FORM LINE FIRST)

    OPTIONS PROMPT LINE 13
    LET answer = ""
    PROMPT " Press any key to continue..." FOR CHAR answer
    CLOSE WINDOW w_win2

# The following options are introduced to illustrate what happens if a ring
# menu does not fit the screen or window.

COMMAND KEY(F5) "Disabling (x)"
        "Disables the close button. Press F5 to activate."
```



```
# The following statement prevents the user from closing the application
# by pressing the X button in the upper right-hand corner of the window.
# Instead, when that button is pressed, a hidden menu command will be
# activated F1 was specified as its activation key. This will be in effect
# until another ON CLOSE APPLICATION statement is encountered. Now you can
# close the application only by selecting the Exit menu option

OPTIONS
ON CLOSE APPLICATION KEY "F1"
CALL fgl_winmessage("Deactivating (x)",
                     "The Close button (x) has been deactivated", "info")

COMMAND KEY(F6) "Enabling (x)"
                 "Enables the close button. Press F6 to activate."
OPTIONS
ON CLOSE APPLICATION STOP

CALL fgl_winmessage("Enabling (x)",
                     "The Close button (x) will now close application", "info")

COMMAND KEY(F7) "Prompt Dialog"
                 "The prompt dialog is used to select an option. Press F7 to activate."

LABEL choice:
LET winpr_answer = fgl_winprompt(2,2,
                                  "Type the option to be highlighted next",
                                  "",16,0)

CASE
WHEN winpr_answer MATCHES "Prompt"
      NEXT OPTION "Prompt"
WHEN winpr_answer MATCHES "Submenu"
      NEXT OPTION "Submenu"
WHEN winpr_answer MATCHES "Windowed Menu"
      NEXT OPTION "Windowed Menu"
WHEN winpr_answer MATCHES "Form Call"
      NEXT OPTION "Form call"
WHEN winpr_answer MATCHES "Disabling (x)"
      NEXT OPTION "Disabling (x)"
WHEN winpr_answer MATCHES "Enabling (x)"
      NEXT OPTION "Enabling (x)"
WHEN winpr_answer MATCHES "Prompt Dialog"
      NEXT OPTION "Prompt Dialog"
WHEN winpr_answer MATCHES "Entering Next"
      NEXT OPTION "Entering Next"
WHEN winpr_answer MATCHES "Selecting Next"
      NEXT OPTION "Selecting Next"
WHEN winpr_answer MATCHES "Next Menu"
      NEXT OPTION "Next Menu"
WHEN winpr_answer MATCHES "Exit"
      NEXT OPTION "Exit"
WHEN winpr_answer IS NULL
      CALL fgl_winmessage("Enter option",
                           "You must enter the option to highlight",
```



```
        "exclamation")
    GOTO choice
OTHERWISE
    LET but_pressed = fgl_winquestion( "Warning",
    "You entered invalid option (they are case sensitive).\nTry again?", 
    "No", "Yes|No", "exclamation", 1)
        IF but_pressed MATCHES "Yes" THEN
            GOTO choice
        ELSE
            EXIT CASE
        END IF
    END CASE

COMMAND KEY(F8) "Entering Next",
    "Selecting next menu option to highlight. Press F8 to activate."

PROMPT "Type name of the option to highlight: "
FOR answer_long -- we use a VARCHAR variable here, because
    -- its size defines the size of the prompt field
    -- for a STRING variable the prompt size would be 1
    -- and the input will be hard to see

BEFORE PROMPT
LET but_pressed = fgl_winquestion( "Activating option",
    "Do you want to activate this menu option?", 
    "Yes", "Yes|No", "question", 1)
IF but_pressed MATCHES "Yes" THEN
    CONTINUE PROMPT
ELSE
    EXIT PROMPT
END IF

AFTER PROMPT -- executed after a value is entered

CASE
    # as we use a VARCHAR variable here,
    # we do not need to clip the value contained
    WHEN answer_long MATCHES "Prompt"
        NEXT OPTION "Prompt"
    WHEN answer_long MATCHES "Submenu"
        NEXT OPTION "Submenu"
    WHEN answer_long MATCHES "Yes"
        NEXT OPTION "Menu_option8"
    WHEN answer_long MATCHES "Windowed Menu"
        NEXT OPTION "Windowed Menu"
    WHEN answer_long MATCHES "Form Call"
        NEXT OPTION "Form call"
    WHEN answer_long MATCHES "Disabling (x)"
        NEXT OPTION "Disabling (x)"
    WHEN answer_long MATCHES "Enabling (x)"
        NEXT OPTION "Enabling (x)"
    WHEN answer_long MATCHES "Entering Next"
        NEXT OPTION "Entering Next"
    WHEN answer_long CLIPPED MATCHES "Selecting Next"
```



```
        NEXT OPTION "Selecting Next"
WHEN answer_long MATCHES "Next Menu"
        NEXT OPTION "Next Menu"
        WHEN answer_long MATCHES "Exit"
        NEXT OPTION "Exit"
OTHERWISE
        CALL fgl_winmessage("No Option",
            "The entered value is invalid",
            "exclamation")
        NEXT OPTION "Entering Next"
END CASE
END PROMPT

COMMAND KEY(F9) "Selecting Next"
"Selecting the next option from a dialog. Press F9 to activate"

LET but_pressed = fgl_winbutton ("Next Option",
    "Which option do you want to highlight next?",
    "Prompt", "Prompt|Submenu|Windowed Menu|Form Call",
    "question",1)

CASE
    WHEN but_pressed MATCHES "Prompt"
        NEXT OPTION "Prompt"
    WHEN but_pressed MATCHES "Submenu"
        NEXT OPTION "Submenu"
    WHEN but_pressed MATCHES "Windowed Menu"
        NEXT OPTION "Windowed Menu"
    WHEN but_pressed MATCHES "Form Call"
        NEXT OPTION "Form Call"
OTHERWISE
        CALL fgl_winmessage("Selection",
            "The menu focus wasn't changed", "info")
        NEXT OPTION "Selecting Next"
END CASE

COMMAND "Next Menu" "Exit from this menu and continue to the next one."

LABEL prompt_rep:
PROMPT "Do you want to move to the next menu? (y/n)" FOR CHAR answer

IF answer MATCHES "[Yy]" THEN
    EXIT MENU          -- this option exits from the current menu
                      -- and 4GL executes the statements below
ELSE
    IF answer MATCHES "[Nn]" THEN
        CONTINUE MENU
    ELSE
        CALL fgl_winmessage("Yes or No",
            "Enter ""y"" or ""n"""", "info")
        GOTO prompt_rep
    END IF
END IF

COMMAND "Exit" "Exit the program" -- after selecting this option the
```



```
-- program will terminate

CLEAR SCREEN
EXIT PROGRAM

# Hidden option activates when (x) is pressed
# after the ON CLOSE APPLICATION option was activated
COMMAND KEY (F1)
CALL fgl_wimmessage("Closing application",
    "Closing application has been disabled, \n use Exit option. ",
    "exclamation")

END MENU

# This statement will prevent the user from closing the application. It
# overrides the OPTIONS ON CLOSE APPLICATION statement with an activation key
# we specified earlier in the program. The only way to close the application
# remains selecting the Exit menu option, but no message is issued.

OPTIONS
ON CLOSE APPLICATION CONTINUE
# Here begins the second menu which executes when you exit from the
# previous one

DISPLAY "Here is an example of a menu with hidden and invisible options:"
    AT 1,2 ATTRIBUTE(GREEN, BOLD)
DISPLAY "----- Press CTRL-W for Help -----"
    AT 4,1 ATTRIBUTE(YELLOW, BOLD)
DISPLAY "Press ! or h to activate the invisible menu options"
    AT 20, 2 ATTRIBUTE(RED, BOLD)

MENU "Second MENU"

COMMAND "Add" "Add a row to the table. Press CONTROL-W to see help." HELP 1
        -- we assign a help message to the menu option
DISPLAY "Adding a row to the table."
    AT 5,2 ATTRIBUTE (GREEN, BOLD)

COMMAND "Change" "Update a row in the table. Press CONTROL-W to see help."
    HELP 2

DISPLAY "Updating a row in the table."
    AT 5,2 ATTRIBUTE (YELLOW, BOLD)

COMMAND "Delete" "Delete a row from the table. Press CONTROL-W to see help."
    HELP 3

DISPLAY "Deleting a row from the table."
    AT 5,2 ATTRIBUTE (RED, BOLD)

COMMAND KEY ("!")-- This is a hidden option without name and description
        -- It can be activated only by the activation key "!"
DISPLAY "You pressed key ""!"". This is the first hidden option."
    AT 5,2 ATTRIBUTE (CYAN, BOLD)
```



```
COMMAND KEY ("h") -- This is a hidden option without name and description
                    -- It can be activated only by the activation key
                    -- "h" or "H"
DISPLAY "You pressed key ""h"". This is the second hidden option.      "
        AT 5,2 ATTRIBUTE (CYAN, BOLD)

COMMAND "Short_menu" "Hides some menu options" -- this command hides
                    -- several menu options:

HIDE OPTION ALL                                -- first it hides all the
                                                -- options
SHOW OPTION "Add", "Long_menu", "Exit" -- then redisplays only
                                                -- three of them

COMMAND "Long_menu" "Redisplays all menu options"
SHOW OPTION ALL                                -- this command is used to display all
                                                -- menu options which were hidden

COMMAND "Exit" "Exit from menu. Press CONTROL-W to see help." HELP 4
        EXIT MENU

END MENU
CLEAR SCREEN

END MAIN
```

The Form File

Below is the form "form_1.per" which is opened by the "Form call" option of the first menu, but you can use any other form for this purpose provided that you refer to the correct form name in the source code file.

```
DATABASE formonly
SCREEN
{
\gp-----q\g
\g|\g form1\g
\g|          |\g
\g|          |\g
\g|          |\g
\g|          |\g
\g|          |\g
\g|          |\g
\g|          [ f ] |\g
\gb-----d\g
}
END

ATTRIBUTES
f = formonly.field;
END
```



```
INSTRUCTIONS DELIMITERS " " "
END
```

The Help File

This application also uses a help file to display the help messages for some menu options. The contents of the help file can vary depending on the information you want to display. The help file for this application must be called "my_help" and it must be compiled. Do not forget that the application references the compiled file and the path to the help file must be that of its compiled version (.erm) and not the source version (.msg).

The "my_help" file used for this application must contain four help messages: .1, .2, .3, and .4. Here is an example of the contents for this file:

```
.1
Do not forget that the form file referenced by the OPTIONS statement must have
.erm extension.

.2
The invisible menu options (which have no name and description) cannot have the
HELP clause.

.3
CONTROL-W key is the default help key. However, you can specify other key or key
combination as the help key.
This can be a function key (F1, F2, F3...) etc.

.4
When you exit a menu, 4GL continues executing the statements following the END
MENU keywords.
Thus the second menu was opened when you pressed "Next MENU" option in the first
one,
though no special code was used to transfer program control to the next menu
explicitly.
The second menu was executed due to the normal order of the program execution.
The "Exit" option of the first menu terminates the program not allowing the 4GL
to proceed to the next menu.
This "Exit" option exits the menu, but as the END MENU keywords are followed by
the END MAIN keywords, the program is nevertheless terminated.
```

Image Files

This application uses several images as the icons for the menu options. These images need to be added to the program requirements. Be careful when naming them and pay attention to the location in which they are located in your project - you may need to change the path referenced by the fgl_setkeylabel() functions in the source code.

-  - new01.ico
-  - edit01.ico
-  - delete01.ico



 - exit01.ico



Interactive Form Elements

We have already discussed [4GL screen forms](#) and the purposes they are used for. The screen forms introduced until now were not interactive, they just displayed some objects created during the form file development. These objects could not be changed at runtime and you could not display values of variables to a form.

4GL has tools which allow you to create interactive objects on a form. These objects can be used to display different values at runtime and to receive the input from a user. This chapter will introduce some of these interactive form objects. It will also explain how you can display something to these form objects. The user input will be discussed later in this manual.

Form Fields

The most often used interactive form objects are form fields. Form fields usually have the form of a limited area within a window or 4GL Screen where the display or input is performed. They are usually one line wide but there are some fields which can take more than one line. Form fields should be specified in the SCREEN section, but they also require several additional form file sections which are not necessary for the non-interactive form objects. We have mentioned the form fields in the 4GL screen forms chapter, now we will discuss them in details.

The SCREEN Section

Form fields as well as other form elements must be added to the SCREEN section of a form. It is an obligatory section of the form file limited by the braces which contains all objects visible on the form. Here is what a form field added to the SCREEN section looks like:

```
SCREEN{  
  
    [ f001 ]  
  
}
```

The SCREEN section above contains one field with tag *f001*. The square brackets represent the beginning and the end of the field, each pair of square brackets in the SCREEN section with at least one character between them is considered to be a field.

The Field Tag

Each field must have a unique name called field tag, it must be written between the delimiter symbols which specify the left and right limits of the field. The field tag must not contain whitespaces, no other printable characters than those specifying the field tag are allowed. You can add whitespaces following the field name, if you want the field to be longer than its tag. The length of a field is equal to the sum total of all the characters present within the field delimiters - its tag and all the trailing white spaces. Thus the field in the example above is 10 characters long and it can accept values consisting of 10 characters. Values containing more than 10 characters will be truncated on the right when displayed to such field. However, this will not affect the actual value of the displayed variable which will remain untruncated.



Field Delimiters

The square brackets are used as the field delimiters in a form. They determine the length of the field and its position within the screen layout. Both delimiters must appear on the same line. If you have two or more fields which you want position close to each other on one line, you can use a pipe (|) symbol between them. The pipe symbol replaces the right bracket of the first field and the left bracket of the second one. Such fields will be displayed next to each other separated only by the pipe symbol. However, doing this requires some further modifications within the form file, otherwise it will not compile. The details are discussed below in the INSTRUCTIONS section.

```
SCREEN{
    [ f001    | f002    ]
    [ f003        ]
}
```

The ATTRIBUTES Section

To be able to add fields to a form file, you should add some more sections following the SCREEN section. The ATTRIBUTES section contains field descriptions that associate field names and data types with field tags in the SCREEN section. Here you also control the behaviour and the appearance of the form fields, supply the default value, limit the range of available values, etc.

The ATTRIBUTES section begins with the ATTRIBUTES keyword and ends at the point where the keyword of another section is specified or where the optional END keyword is placed. For now all our fields are formonly fields. This means that you need use the DATABASE formonly section before the SCREEN section. The formonly fields are not linked to any database and all their attributes are declared in the ATTRIBUTES section. Here you should declare that the field is a formonly field and assign a name to it. The simplest field description contains the field tag and its name preceded by the formonly prefix:

```
ATTRIBUTES
field_tag = formonly.field_name;
```

The field tag is the name of the field you have specified in the SCREEN section. The field name can then be used in the 4GL source code to address this field. The field name and field tag can match. The description of each field must be followed by a semicolon.

Here is an example of the ATTRIBUTES section for three formonly fields:

```
DATABASE formonly
SCREEN{
    [ f001        ]
    [ f002    ]    [ f003    ]

}
ATTRIBUTES
f001 = formonly.f001;
f002 = formonly.my_field;
```



```
f002 = formonly.f003;
```

Data Type Specification

You can specify a data type for a field. It will restrict the values which can be displayed or input in such field to the specified data type and the compatible data types. If you do not specify the data type (all the examples above in this chapter do not have a data type specified), the field is by default of the CHAR data type, which is more or less compatible with the rest of the simple data types.

To specify the data type of a field use the TYPE keyword after the field name in the ATTRIBUTES section.
E.g.:

```
ATTRIBUTES
f001 = formonly.f001 TYPE MONEY;
f002 = formonly.f002 TYPE DATE;
```

Do not assign a length to CHAR, DECIMAL, and MONEY fields because field length is determined by the field width in the SCREEN section. Note that a field can be only of simple data type. This means of one of the data types we have already discussed in the previous chapters. All the data types which will appear later in this manual cannot be assigned to a form field.

The INSTRUCTIONS Section

Another form section which becomes obligatory when you include fields into your form is the INSTRUCTIONS section. This is a final section of the form specification file, no other sections should follow it. This section can contain several clauses. For now we are only interested in the DELIMITERS clause.

The DELIMITERS clause specifies what delimiters are used for form fields when the form is displayed on the screen. The DELIMITERS keyword must be followed by the opening and closing delimiter symbols enclosed within quotation marks (").

```
INSTRUCTIONS
```

```
DELIMITERS "[ ]"
```

Each delimiter occupies a space, so two fields on the same line are usually separated by at least two spaces. To specify only one space between consecutive screen fields, use the pipe (|) symbol in the SCREEN section. To be able to use it, follow this procedure:

1. In the SCREEN section, substitute a pipe symbol (|) for paired back-to-back (][) brackets that separate adjacent fields.
2. In the INSTRUCTIONS section, define the same symbol as both the beginning and the ending delimiter. You cannot specify brackets as the delimiter in such case, because these are two different symbols. For example, you can specify "\\", or "||", or "::".



The DELIMITERS section only influences the appearance of the fields at runtime. When adding them to the form file you should still use the brackets and pipe as the delimiters. If you use any other symbols as the delimiters in the SCREEN section, they will be treated as literal symbols and not as a field.

Here is an example of a complete form specification file called "fields_form.per" with two form fields and some non-interactive form elements:

```
DATABASE formonly
SCREEN{
  \g-----
    No.1      No.2
    [f001      |f002      ]
  \g-----
}
ATTRIBUTES
f001=formonly.f001 TYPE INT;
f002=formonly.second_f;
INSTRUCTIONS
  DELIMITERS ":" :
```

At runtime, the above example will be displayed as follows:

No.1	No.2
:	:

Thus you see that the delimiters used in the DELIMITERS section do not influence the ones used in the SCREEN section, but they do influence the result during the runtime.

The delimiters specified in the DELIMITERS section do not influence the fields representation in a GUI mode. The fields edges will look the same whatever symbols you specify as the delimiters in the form file. However, the fields will be located closer to each other than if no pipe symbol is used. Therefore, the same form will look as follows when the application is run in Phoenix:

A screenshot of a Phoenix application window. It contains two input fields side-by-side. Above the first field is the label "No.1" in blue. Above the second field is the label "No.2" in blue. The input fields are rectangular boxes with a light gray background and a thin black border. There is a horizontal line above the fields and another below them, separating them from the labels.



	<p>Note: To make 4GL display a form file you should follow the instructions given in "4GL Form Files" chapter.</p>
---	---

Multi-Segment Fields

In 4GL a field can occupy more than one line. Such fields are called multi-segment fields and they are used to input or display the amount of information which cannot fit one line.

A multi segment field is specified in the SCREEN section with the help of a number of fields placed one under another which have the same field tag. The example below contains a single form field with three segments:

```
SCREEN{  
    [f001] [f001] [f001]  
}
```

In the ATTRIBUTES section such kind of field is declared only once regardless of the number of sections. The special WORDWRAP attribute is used to instruct 4GL that this field is a multi-segment one. E.g.:

```
ATTRIBUTES  
    f001=formonly.f001, WORDWRAP
```

If a character string is displayed to such field, it will be continued onto the lower segments, if it does not fit one line.



Displaying Data to Fields

Now as you know how to create fields and display them at runtime, we will touch upon the issue of displaying information to form fields. It can be done using the DISPLAY statement we already know. The DISPLAY statement can display information onto a line in a window or 4GL screen and into form fields. The syntax of the DISPLAY statement which performs output to form fields differs from the syntax we know so far.

The syntax of a DISPLAY statement displaying data to a form field is as follows:

```
DISPLAY value TO field_name
```



The *value* here is either a literal value (a number, a quoted character string, etc.) or a variable which value is to be displayed to the field specified by the *field_name*. Note that the *field_name* is not the same as the field tag you use in the SCREEN section. You can assign the field name to a variable and use this variable in the TO clause. If you use the literal form filed name, do not enclose it in quotation marks. The field name is assigned to a field tag in the ATTRIBUTES section, for the details see the "ATTRIBUTES section" above. You can display several values to several fields using one DISPLAY statement. The values and field names should be separated by commas and they should match in number and order.

Here is the code which displays values to the form fields from the sample from file on the previous page:

```
OPEN WINDOW my_win AT 2,2 WITH FORM "form_fields"  
DISPLAY 1234, "value 2" TO f001, second_f
```

Here is what it will look like at runtime when the application is run in the character and in the GUI modes:



Additional Field Attributes

A form field can have additional characteristics such as colour or comments. All these characteristics are adjusted using the ATTRIBUTES section. One field can have a number of different attributes. They must be separated by commas. The additional attributes must also be separated by a comma from the field name and optional TYPE declaration.

Below we will discuss the colour attributes and the display conditions. There can be other attributes. However, they will be introduced later in the following chapters of this manual, because most of them take effect when a field is used for input.

Colour Attributes

The COLOR attribute displays field contents in a colour or with other video attributes. It has the following syntax:

```
COLOR = attribute [attribute]
```

The *attribute* here is any of the standard attributes described previously for the [DISPLAY statement](#) (excluding the INVISIBLE attribute), they follow the same rules as the standard display attributes, thus more than one colour attribute cannot be used in one and the same WHERE clause:

- Colour attributes: BLACK, BLUE, CYAN, GREEN, MAGENTA, RED, WHITE, and YELLOW
- Intensity attributes: REVERSE, BLINK, LEFT, and UNDERLINE



If you have more than one attribute following the COLOR keyword, you should **not** separate them by commas.

If we apply colour attribute to the previously used form file, it will look as follows:

...

ATTRIBUTES

```
f001=formonly.f001 TYPE INT, COLOR=REVERSE;
```

```
f002=formonly.second_f, COLOR=GREEN;
```

...

If we then use the same code to open the form and display the values as in the "Displaying data to fields" section, we will receive the following result:

No.1	No.2
: 1234	: value 2

No.1	No.2
1234	value 2

Conditioned Colour Attributes

You can use colour attributes depending on the value displayed in a field. Thus you can use the optional WHERE clause of the COLOR attribute to specify the condition which should be met for the attribute to take effect.

The condition consists of the WHERE keyword and a Boolean expression following it, the WHERE clause should **not** be separated from the colour attribute by a comma. The Boolean expression used in the WHERE clause can contain the field tag, this needs to be the field tag of the field you are creating the WHERE clause for. The typical formats of the WHERE clause are listed below:

- *field tag | relative operator | 4GL expression or field tag* - such expression compares the content of the field specified by the first field tag with the result of some expression, or with the contents of another field. E.g.:

```
...COLOR = RED WHERE f001>1...
...COLOR = BLUE REVERSE WHERE f002<>f003
```

- *field tag IS [NOT] NULL* - such expression specifies whether the field can contain NULL values for the attribute to take effect. E.g.:

```
...COLOR = BLINK WHERE f001 IS NOT NULL...
```

- *field tag [NOT] BETWEEN 4GL expression or field tag AND 4GL expression or field tag* - such expression checks whether the field contents belongs to the specified range. E.g.:



```
...COLOR = REVERSE WHERE f003 BETWEEN f001 AND f010...
```

- *(field tag) [NOT] IN (4GL expression list)* - this expression checks whether the value in the specified field matches any value specified in the parentheses. The values in the parentheses must be separated by commas, if they are non-consecutive, and can have a dash between them if you want to specify a consecutive set of possible values. E.g.:

```
...COLOR = YELLOW BLINK WHERE f001 IN(10-100)...
```

The IN() Operator

The IN() operator is valid in the WHERE clause of the COLOR attribute and in some other cases which will be discussed later in this manual. It is not valid in normal 4GL expressions. This operator is used to perform a membership test. It has the following syntax:

```
field_tag [NOT] IN(expression_list)
```

It checks whether contents of the field matches any of the values specified by the IN() operator. It returns TRUE or false depending on whether or not the NOT keyword is included. The following example returns TRUE, if the f001 field contains numbers from 1 to 5 and FALSE if it contains any other data:

```
... WHERE f001 IN(1,2,3,4,5)...
```

The following example returns FALSE under the same condition:

```
... WHERE f001 NOT IN(1,2,3,4,5)...
```

The BETWEEN... AND Operator

The BETWEEN ... AND operator checks whether the field contents is within the range specified by it. Here is its syntax:

```
field_tag [NOT] BETWEEN expression AND expression
```

The contents of the field and the both operands of this operator must be of compatible data types. The first operand must be smaller than the second one. If they are date values, the first must be earlier than the second one. If they are intervals, the first must be smaller than the second.

If you omit the NOT keyword, this test evaluates as TRUE if the field contents is not less than the second operand or greater than the third. If you include the NOT keyword, the test evaluates as FALSE if the field contents is a value outside the specified range.

A Complex WHERE Clause

You can also use more than one Boolean expression in one WHERE clause, if you specify either AND or OR operator between them. In such case the conditions will follow the general rules for the AND and OR operators. The Boolean and relative operators and expressions were discussed earlier in "[Conditional Statements and Logical Expressions](#)" chapter.

Here is a previously used example, but the COLOR attributes now have conditions:



```
...
ATTRIBUTES
f001=formonly.f001 TYPE INT,
    COLOUR=REVERSE WHERE f001<1 AND NOT IN(1233, 1234, 1235);
f002=formonly.second_f, COLOUR=GREEN WHERE f002="value 2";
```

Such conditions will cause 4GL to apply the colour attributes to field f002 because value displayed to it is exactly "value 2". However, the colour attribute will not be applied to field f001. Though the first part of the WHERE condition is TRUE and the value displayed is greater than 1, the second part is FALSE, so the attribute is not applied.

Other Widgets

Except the field and the dynamic label there are a number of other widgets. In this chapter we discuss the widgets, which cannot be used for input and which typically serve for displaying some information or providing the access to the file system or to the Internet.

The Dynamic Label

A dynamic label is a simple form label which has a static location but dynamic content. You should not confuse dynamic labels with the text displayed by the DISPLAY AT statement or static form labels. You can use the DISPLAY TO *field* statement to display some data to a dynamic label. Since it is still a label, it cannot be used for input unlike fields.

You should use dynamic labels, rather than static ones and DISPLAY AT statement, to display any text to the form. Dynamic labels are more convenient to use than static ones and provide you with a number of useful features. For example, you can specify the font attributes such as colour, alignment, and intensity for the text of a dynamic label; you can also reference a dynamic label as any other form field.

Another helpful feature of dynamic labels is that they use modern proportional fonts; it means that the displayed string can be longer than the declared length of the dynamic label. If a string is too large and does not fit the dynamic label length, it will be clipped. This prevents other form objects from being overlapped by the content of the dynamic label.

By default, the text displayed to the dynamic label is left aligned, and the numeric values are right-aligned. You can change the default alignment using the LEFT and RIGHT attributes in the ATTRIBUTES section. The field declaration requires the "config" attribute which is often used in different widget types which will be discussed in the following chapters. This attribute accepts widget specific values. In the case with the dynamic label it specifies the initial text to be displayed to the label. And as any other form widget except simple fields it requires the "widget" attribute which tells 4GL what kind of widget it is.

The Attributes syntax for a dynamic label widget is:

```
field_tag = database.field_name,
config="The string displayed initially when the form is displayed",
widget = "label" -- this keyword denotes a dynamic label widget
```



When you add a dynamic label to the SCREEN section, you must put the field tag within the square brackets, as you do it with a usual form field. All other form widgets also look like normal fields in the SCREEN section, though they differ in their declaration in the ATTRIBUTES section.

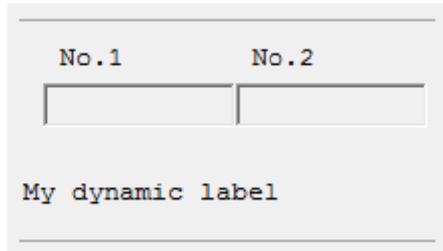
You can add a dynamic label tagged as "f003" to the form that we have already created:

```
SCREEN{
  \g-----
    No.1      No.2
  [f001      |f002      ]
  [f003      ]
  \g-----\g
}
```

The attributes specification for the f003 field will be as follows:

```
ATTRIBUTES
f001=formonly.f001 TYPE INT;
f002=formonly.second_f;
f003=formonly.labl, config = "My dynamic label", widget = "label";
```

When you add these lines to your form specification file and run the application, you will see the following in the screen (in character and in GUI modes):



The Browser

The browser widget is a form field which lets you add a fully functional browser into the screen form. Phoenix uses the Internet explorer browser, and Chimera uses the embedded browser provided with Java runtime environment. The browser widget is not fully supported when the application is run in the character mode. If you launch your application in this mode, the browser fields will remain empty.

The file browser will try to connect to the source specified as the browser field value. This can be an HTML file, an image, any other files located on the system where the application is run which the browser supports. It can also be an URL of a web page, but it requires Internet connection. The file browser can also be useful in merging 4GL applications with Web-based applications.

You should follow the next syntax when specifying a browser widget:

```
field_tag = database.field_name,
config = "url or file system path",
```

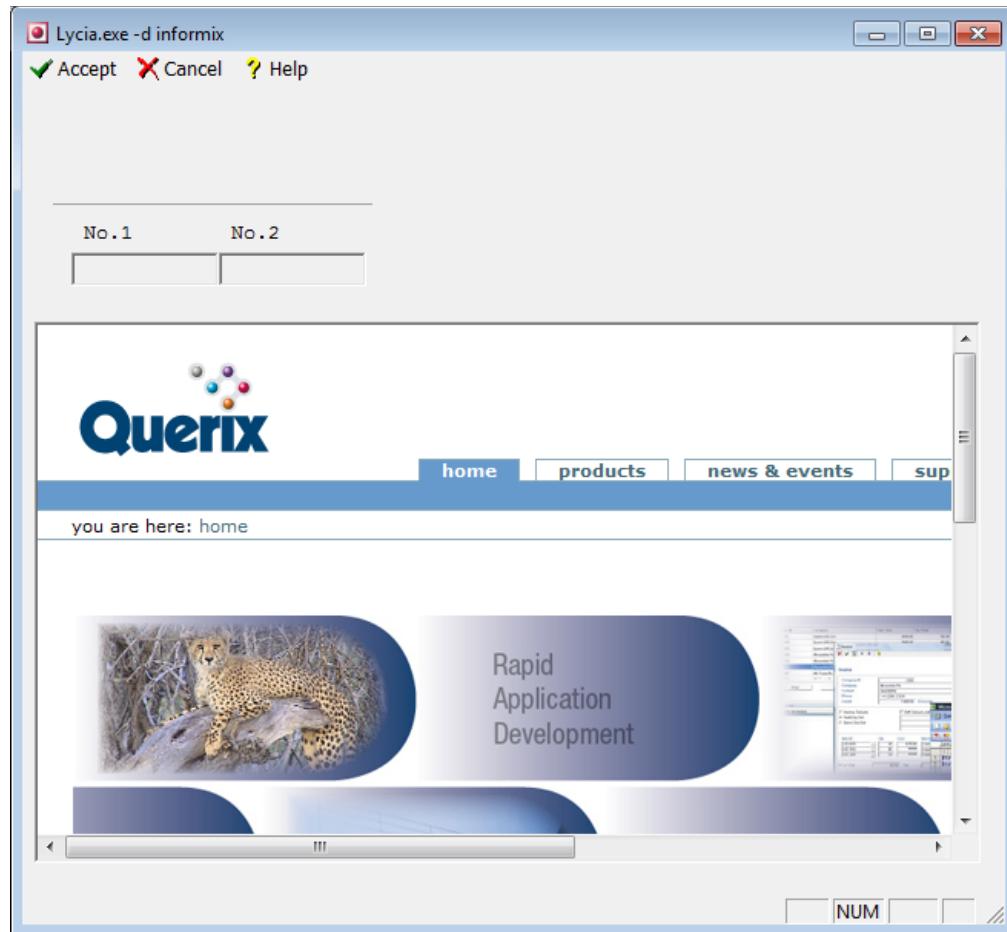


```
widget = "browser"
```

Here is an example of the code for the field tagged f003 which we want to become a browser:

The document or the HTML page will open directly in the field specified as a browser. So, for the sake of security, if you click on the link, it will open in the browser. If the browser field is empty, it will open in the default browser.

Such field specification in combination with the ATTRIBUTES given above will result in the following application message:



The URL is taken from the config attribute value, therefore, the value can be changed dynamically at any time by means of statements such as DISPLAY... TO <field>. Thus here is how you can override the URL specified in the form file, if you specify this line right after the form file is displayed, or change it at some point of the program execution:

```
DISPLAY "www.google.com" TO formonly.mybrs
```

You can also specify the *config* property as "c://" - this will open the root directory of the disc C; or "c:/Images/my_im.jpg" - this will display a picture named my_im, stored in the folder *Images* on the disc C.

The Image

The image widget is used to create a static image area. You can initialize the image path in the form file and change it at runtime. This image is non-clickable and is displayed with a 1:1 scale.

The image widget supports BMP, JPG, TIF, ICO, PNG, and GIF formats. If you run the program in Character mode, any image field will be displayed as a normal field.

To specify an image widget, you should follow the attributes syntax given below:

```
Field_tag=database.field_name,
```

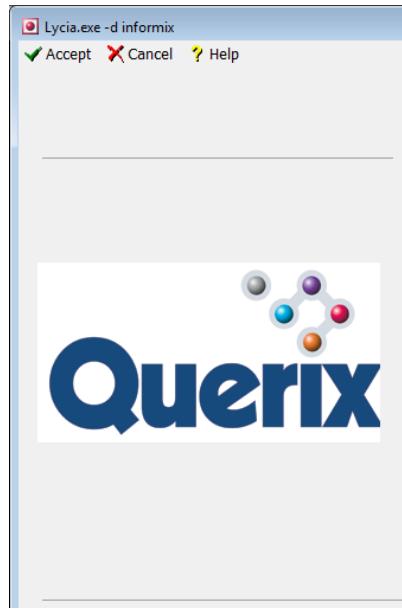


```
config = "image_path",  
widget = "image";
```

If a file *querix_logo.png* is located at the same folder with the executable file, we can specify the f003 field as an image widget in the following way:

```
f003=formonly.myimg,  
config = "querix_logo.png",  
widget = "image";
```

If it is the only form field, the form, when displayed, will look as follows:



You can change the image at runtime by means of the DISPLAY TO statement. To do it, specify the image path in quotes after the DISPLAY keyword and the image field name after the TO keyword:

```
DISPLAY "images/logo.jpg" TO myimg
```

By default, all the images are enabled, but you can disable an image and make it invisible. To do this, use the DISPLAY "*" TO *field_name*. To enable the image that was earlier disabled, use the DISPLAY "!" TO *field_name* structure.

Text Alignment

For almost all the widgets it is possible to specify the alignment according to which the information will be displayed. The alignment can be specified along with other attributes. There are three keywords which specify the alignment:

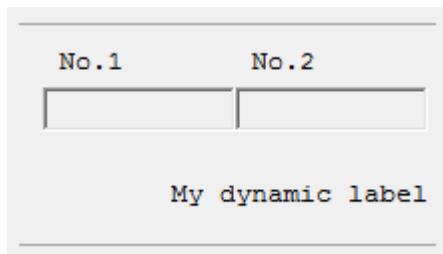


- **RIGHT** - aligns the text to the right side of the widget (default for numeric values);
- **LEFT** - aligns the text to the left side of the widget (default for character values);
- **CENTER** - centres the text.

The example below applies the **RIGHT** attribute to a dynamic label:

```
f003=formonly.labl, RIGHT, config = "My dynamic label", widget = "label";
```

In this example, the text in the dynamic label is displayed right-aligned:



A Dialog Box for Manipulating Files

The *fgl_file_dialog()* function creates a dialog box which allows the user to save or to open a file. When the user chooses the file, the function returns a character string, indicating the path to the file, the file name and extension.

The function can include from two up to six parameters, which are shown in the following scheme. The parameters taken in brackets are optional, the other two are obligatory, their position should be exactly as in the syntax scheme below:

```
CALL fgl_file_dialog (type, multiselect [, "title"] [, "default_path"]
[ , "default_filename"] [, "format (*.extention)"] )
```

- **type** stands for the type of the dialog box you want to call, can be: *"open"* or *"save"* (the quotation marks are obligatory).
- **multiselect** option specifies the number of files that the user can select. The value can be *1* which means that only one file can be selected at a time, or *0* which allows the user to select several files.
- **title** is used to specify the title for the dialog box, it can be a quoted character string.
- **default_path** specifies the default filename which will be automatically inserted to the dialog box's filename field.
- **default_filename** refers to the default filename that is stored in the dialog box's filename field.
- **format** option consists of the format name and the file extension which serves as the filter for the files to be displayed in the dialog box.



	<p>Note: This function can be executed only if you run the application in the GUI mode</p>
---	---

The following example demonstrates the *fgl_file_dialog()* function with all the options included:

```
CALL fgl_file_dialog ("open", "1", "Please select a file to open",
"defaultpath", "my_file", "any format")
```

However, such function invocation will have no result, because the program won't fix the user's choice. When the following program is run, the user is to press the F2 key if they want to add a photo to the form with their personal data:

```
MAIN

DEFINE data RECORD
    path VARCHAR(500),
    fname, lname, city CHAR(20),
    age INT
END RECORD

OPEN WINDOW w1 AT 2,2 WITH FORM "myf"
ATTRIBUTES (BORDER)

#Inputs first and last name and age
INPUT data.fname, data.lname, data.age FROM pers.fname, pers.lname,
pers.age

BEFORE INPUT
    Message "Press F2 to download a photo"

# Opens a dialog box if the user presses F2 and lets them choose a file
ON KEY ("F2")

    LET data.path = fgl_file_dialog ("open", "1", "Please select a file to
open", "defaultpath",
"my_file", "Picture (jpeg) | *.jpg | " ) -- JPG files are allowed
```

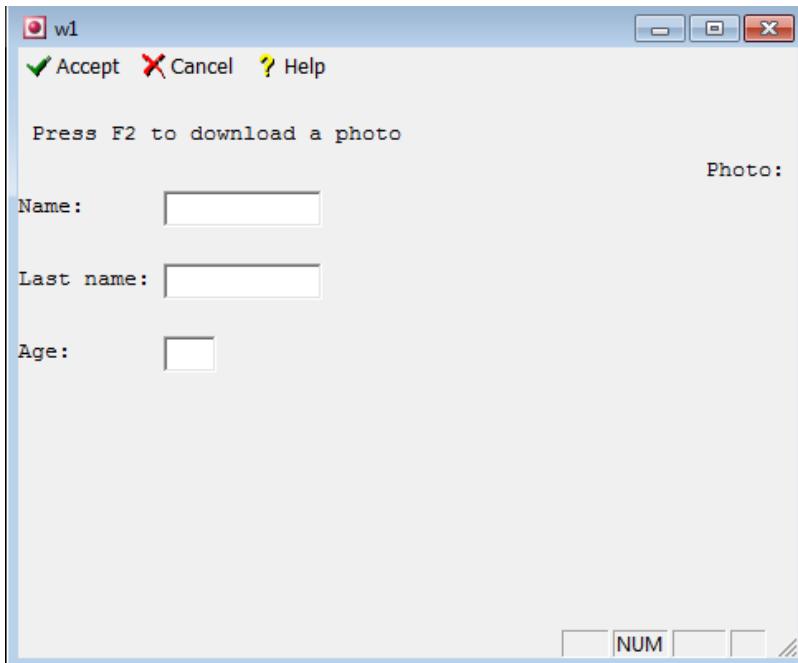


```
#displays the chosen file to the form
DISPLAY path TO pers.photo

END INPUT

END MAIN
```

When the program is run, the user can see the following:



When they press the F2 key, the dialog box opens. Select the file you want to open and click **Open**.

After the user chooses the necessary file and clicks the button "Open", the path to the file is stored to the variable *path* and after that, this variable is used to specify the picture displayed to the image widget named *photo*. As the result, the chosen picture becomes displayed to the form:



Example

The following example consists of several source files - one 4GL file and two forms. It shows how information can be displayed to form.

```
#####
# This example illustrates the basics of the field management in a screen form
#####
MAIN
# Below we define variables which then will be displayed to the form
DEFINE
    prom      CHAR,
    i         INTEGER,
    id        INTEGER,
    first_name,   CHAR(30),
    last_name,   DATE,
    date_birth,  INTEGER,
    phone,      CHAR(14),
    units_pd,    DECIMAL(5,2),
    price_pu,    MONEY(16,2),
    id_card,    CHAR(10),
    date_issue,  DATE,
    date_expire, VARCHAR(80),
    address,    CHAR(20),
    city,       CHAR(6),
    zip_code,   CHAR(20),
    country_name, CHAR(512),
    add_data,   CHAR(512),
    accept_date, DATE,
    start_date, DATE,
```



```
start_time      DATETIME HOUR TO SECOND,
end_date        DATE,
end_time        DATETIME HOUR TO SECOND,
img_lab         VARCHAR (30), -- These are variables
page_lab        VARCHAR (50), -- to store values for
wsite_lab       VARCHAR (60), -- dynamic labels
ans             VARCHAR (70)  -- And this variable will be used to hold
                           -- the values returned by the
                           -- fgl_winquestion()and fgl_winprompt()
                           -- functions

# Here we verify that the application is run in GUI mode
# It uses some form widgets non-displayable in character mode,
# so if it is run in the character mode, it will be closed.

IF fgl_fglgui()= 0 THEN
  CALL fgl_winmessage("Wrong mode",
                      "Run this application in the GUI mode.",
                      "info")
  EXIT PROGRAM
END IF

# Below we assign values to variables which we want to display to the form
# fields
LET id          = 1
LET first_name  = "John"
LET last_name   = "Smith"
LET date_birth  = "03/08/1965"
LET age          = YEAR(TODAY) - YEAR(date_birth)
LET phone        = "0663850322"
LET units_pd    = 5.5
LET price_pu    = 17.23
LET id_card     = "2397213D"
LET date_issue  = "02/01/2000"
LET date_expire = "02/01/2015"
LET address     = "102, Everton Rd."
LET city         = "Liverpool"
LET zip_code    = "123456"
LET country_name= "United Kingdom"

# Here we assign a number of character strings to a variable in order to
# display them to a multi segment field

LET add_data    =
  "Here the additional comments are displayed to a multisegment field. ",
  "This field can comprise several lines of text either displayed to it",
  "or entered by a user. In a form file a multi-segment field looks like",
  "several fields with the same tag placed one under another. ",
  "The WORDWRAP attribute is required to create a multi segment field."
```



```
LET accept_date = TODAY+7 UNITS DAY
LET start_date = TODAY
LET start_time = CURRENT HOUR TO SECOND

LET end_date = "12/31/9999"
LET end_time = "23:59:59"

OPEN WINDOW w_persons AT 2,1 WITH 28 ROWS, 80 COLUMNS ATTRIBUTES (FORM LINE
FIRST)
OPEN FORM appl_form FROM "application_form"
DISPLAY FORM appl_form

# We display the data to the dynamic label. This information will be changed
# throughout the program. This frees us from the necessity to save the line
# free and clean it
DISPLAY "The values of variables are displayed to form fields. Press any
key." TO formonly.label_main

# We display values of variables to fields
# Pay attention to the order of variables and fields - they coincide
DISPLAY id,first_name,last_name,date_birth,age,phone,units_pd, price_pu,id_card,
date_issue,date_expire,address,city,zip_code,country_name,
accept_date,add_data,start_date,start_time,end_date,end_time
TO id,first_name,last_name,date_birth,age,phone,units_pd,
price_pu,id_card, country_name, date_issue, date_expire,
address,city,zip_code, accept_date,add_data,start_date,start_time,
end_date,end_time

CALL fgl_getkey()

DISPLAY "The colour attribute changes depending on the value, press any key" TO
label_main
# Here is the demonstration of different attributes applied to a field,
# depending on the value displayed to it
CALL fgl_getkey()

DISPLAY "For 9.00 it is NORMAL. Press any key to continue." TO label_main
ATTRIBUTE (NORMAL)
LET price_pu = 9.00
DISPLAY price_pu TO price_pu
CALL fgl_getkey()
DISPLAY "For 15.60 it is GREEN. Press any key to continue." TO label_main
ATTRIBUTE (GREEN)
LET price_pu = 15.60
DISPLAY price_pu TO price_pu
CALL fgl_getkey()
DISPLAY "For 23.89 it is YELLOW. Press any key to continue." TO label_main
ATTRIBUTE (YELLOW)
LET price_pu = 23.89
DISPLAY price_pu TO price_pu
CALL fgl_getkey()

CLOSE FORM appl_form
CLOSE WINDOW w_persons
CLEAR SCREEN
```



```
# We open another window with a form in order to demonstrate the use of
# the interactive form elements discussed in the chapter
OPEN WINDOW win2 AT 2,2 WITH FORM "pers_pref" ATTRIBUTES (FORM LINE FIRST)

DISPLAY "The default information is displayed first. Press any key." TO
label_main

# We assign values to the following variables
LET img_lab = "PHOTO"
LET wsite_lab = "FAVOURITE WEB-SITE"
# and display them to the dynamic labels in the form.
DISPLAY img_lab, wsite_lab TO img_lab, wsite_lab
CALL fgl_getkey()

DISPLAY "Now you can change the photo. Press any key to continue." TO
label_main

# The fgl_dialog_box() is invoked and the user is offered to select an image
# file for download. The path to the file and its name are assigned to a
# variable specified in the RETURNING clause. The value of the variable will
# be evaluated in the conditional statement. Note that the size of the image
# must fit the area in the form where it will be displayed. If it doesn't, it
# will overlap some form elements
DISPLAY "*" TO img -- This will hide the default image
CALL fgl_getkey()
CALL fgl_file_dialog("open", "1", "Please select a file to open", "",
                     "my_image", "*.jpg") RETURNING ans
IF ans IS NULL THEN
    DISPLAY "" TO img
    DISPLAY "You've selected no photo." TO label_main
ELSE
    DISPLAY ans TO img
END IF

CALL fgl_getkey()
# Finally, the user is prompted to provide the URL of their favourite web-
# site. The result will be displayed in the browser field.
DISPLAY "Now you can select the web site." TO label_main
LET ans = fgl_winprompt(2,2,"Enter the URL of your favourite web-
site, if any","",30,0)

IF ans IS NULL THEN
    DISPLAY "www.querix.com" TO wsite -- If no value has been
                                         -- entered into the prompt
                                         -- field,
                                         -- the default site main page
                                         -- remains displayed in the
                                         -- browser field
ELSE
    DISPLAY ans TO wsite
END IF

CALL fgl_getkey()
```



END MAIN

Form Files

This is the first form "application_form.per"

```

DATABASE formonly
SCREEN SIZE 26 BY 80

{
\gp----[label_main
\g|\g      ID Number:\g[f00      ]
\g|\g      First Name:\g[f01      ]
\g|\g      Last Name:\g[f02      ]
\g|\g      Birth Date:\g[f05      ]\g Age:\g[f6]
\g|\g      Phone:\g[f07      ]
\g|\g Units per day:\g[f08      ]
\g|\gPrice per unit:\g[f09      ]
\g-----\g
\g|\g ID card:\g[f10      ]\g Issued on:\g[f11      ]\g Expires on:\g[f12      ]
\g-----\g
\g|\gAddress:\g[f13      ]
\g|\g City:\g[f14      ]\g Zip:\g[f15      ]\g Country:\g[f17      ]
\g-----\g
\g|[f18      ]
\g|[f18      ]
\g|[f18      ]
\g|[f18      ]
\g|[f18      ]
\g-----\g
\g|\gRegistered:\g[f19      ][f20      ]\g Dismissed:\g[f21      ][f22      ]
\g|\g Accepted:\g[f23      ]
\gb-----d\g
}

ATTRIBUTES
label_main = formonly.label_main, widget="label";
f00 = formonly.id TYPE INTEGER, LEFT, REVERSE;
f01 = formonly.first_name TYPE CHAR;
f02 = formonly.last_name;
f05 = formonly.date_birth TYPE DATE;
f6 = formonly.age TYPE INTEGER;
f07 = formonly.phone, RIGHT;
f08 = formonly.units_pd TYPE DECIMAL;
f09 = formonly.price_pu TYPE DECIMAL, COLOR = GREEN WHERE f09 >= 10.00
          AND f09 <= 20.00,
          COLOR = YELLOW WHERE f09 > 20.00;
          -- we apply conditional colour attributes for this field
f10 = formonly.id_card TYPE CHAR;
f11 = formonly.date_issue TYPE DATE;
f12 = formonly.date_expire TYPE DATE;
f13 = formonly.address TYPE VARCHAR;
f14 = formonly.city TYPE CHAR, CENTER;
f15 = formonly.zip_code TYPE CHAR;
```



```
f17 = formonly.country_name, CENTER; -- We have changed the text alignment in
-- this field by using attribute
-- CENTER
f18 = formonly.add_data TYPE CHAR,WORDWRAP; -- this is a multi-segment field

f19 = formonly.start_date TYPE DATE;
f20 = formonly.start_time TYPE DATETIME HOUR TO SECOND;
f21 = formonly.end_date TYPE DATE,COLOR = INVISIBLE WHERE f21 = "12/31/9999";
f22 = formonly.end_time TYPE DATETIME HOUR TO SECOND,
          COLOR = INVISIBLE WHERE f22 = "23:59:59";
          -- we apply conditional colour attributes for this field
          -- as the attribute is INVISIBLE, the values will not be seen
f23 = formonly.accept_date TYPE DATE;
```

INSTRUCTIONS

DELIMITERS "[]" -- these are standard delimiters

This is the second form "pers_pref.per"

DATABASE **formonly**

SCREEN SIZE 26 **BY** 80

```
{
  [label_main
    [lab1
      ]
    ]
  \g-----\g
  [f01
    ]
    \gp-----q\g
    \| [f04
    \| [f04
    \| [f04
    \| [f04
    \| [f04
    \| [f04
    \|gb-----d\g
  \g-----\g
  [f03
    ]
  [f06
    ]
  \g-----\g
}
```

ATTRIBUTES

```
label_main = formonly.label_main, LEFT,
            config = "The main descriptive label",
```



```
        widget = "label",
        COLOR = GREEN;
lab1=formonly.lab1, widget="label", CENTER, config="PERSONAL PREFERENCES", COLOR =
CYAN REVERSE;

# Fields f01, f02 and f03 are dynamic labels. They receive values at runtime
f01 = formonly.img_lab, widget="label", config = " ";
f03 = formonly.wsite_lab, WORDWRAP, widget="label", config=" ";
# Field f04 is the image widget. Notice that we have included the WORDWRAP
# attribute to let the image take up several lines in the form. You should
# change the path in the config attribute for the path where the image
# resides on your system. It can be an image imported into the program
# or located anywhere on your file system
f04=formonly.img,WORDWRAP,widget="image", config="default_avatar.jpg";
# f05 is the hotlink widget. We have not included the default URL in this
# case. But the label will be displayed.
# And this is the browser widget with the default URL provided
f06 = formonly.wsite, widget ="browser", config = "www.querix.com";
```

The Image File

This is the image "default_avatar.jpg" the path to which was specified in the config attribute of the image widget. If the image is imported into the program and is located in the same directory as the source files, you need not change the path given above. You can copy it to your file system and change the path to it in the example, or use your own image.





Screen and Program Records

Earlier in this tutorial, we have discussed different simple data types. However, 4GL supports two more classes of data types: structured data types and large data types. In this chapter, we will discuss the RECORD, one of the data types, classified as *structured*.

A Program Record

The RECORD is a structured data type which is represented by an ordered set of variables. The components of a program record (which are also called *record members*) may be a combination of any simple, structured, or large data types.

The order of the record members is fixed, it is specified when a variable of RECORD data type is declared and cannot be changed throughout the program.

The syntax of a RECORD data type declaration is as follows:

```
DEFINE
    record_name RECORD member1 [, member 2, member 3 ...] END RECORD
```

The *record_name* is an identifier by which a program references the record. When you specify the record members, you should follow the usual rules of variable name specification.

RECORD Members

You have to declare a record and its members in a declaration clause. The record declaration starts with the record name followed by the RECORD keyword and ends with the END RECORD keywords. The record members are declared between the RECORD and END RECORD keywords. The declaration of record members is similar to the usual variable declaration:

```
DEFINE

    my_rec RECORD

        mem1_1, mem1_2 INTEGER,
        mem1_3 CHAR ( 20 ),
        mem1_4 VARCHAR(15)

    END RECORD
```

This piece of the source code will create a record named *my_rec* and containing four members: two of the INTEGER data type, one of the CHARACTER data type and one of the VARCHAR data type.



Nested Records

Record members can be of RECORD data type, thus you can create nested records, where one or more members of a record are records of their own. In the example below the "my_rec" record contains another member which is of RECORD data type:

```
DEFINE

my_rec RECORD

    mem1_1, mem1_2 INTEGER,

    mem1_3 CHAR ( 20 ),

    sec_rec RECORD

        mem2_1, mem2_2 DATETIME

    END RECORD,

    mem1_4 VARCHAR(15)

END RECORD
```

You should not forget that each record regardless of whether it is nested or not should end with the END RECORD keywords.

Referencing Record Members

A program may refer to a record as the whole or to separate record members. There are several ways to refer to records and their members:

- To refer to the entire record, use the notation *record.** Where *record* stands for the record name. The asterisk (*) stands for all of the members of the record.
- To refer to a specified member of a record, you can use the structure *record.member*, where *record* stands for the record name, and *member* stands for the member identifier.
- You can also refer to a consecutive subset of record members, using the THRU (or THROUGH) keyword. The syntax is: *record.first THRU record.last* where *first* and *last* stand for the first and the last elements of the members subset.
- To reference a member of a nested record, you need to specify the name of the primary record, then the name of the nested record and then the member of the nested record: *record.nested_record.nested_r_member*. If we look at the example of a nested record above, to reference mem2_1 member you'll need to specify *my_rec.sec_rec.mem2_1*.

When you assign values to members of a record, you can assign all of them by listing them after the LET statement, or you can assign a value to a specified record member by using the structure *record.member*:

```
LET my_rec.mem1_3 = 56 -- the value is assigned to the member
-- named mem1_3
```



```
LET my_rec1.* = 101, 102, 103, 104, 105 -- the values are
-- assigned to all record members
```



Note: If your record has another record as a member, assigning values using `LET my_rec1.* = 101, 102, 103, 104, 105` method will not assign any values to the members of the nested record. To assign values to all the members of a nested record you should do it explicitly, e.g. `LET my_rec1.nest.* = 101, 102, 103, 104, 105`

The following part of a source code will demonstrate how a program record can be declared and referenced:

```
MAIN

DEFINE

    my_rec1 RECORD          --record declaration
        mem_11, mem_12, mem_13, mem_14, mem_15 INTEGER
    END RECORD

    LET my_rec1.* = 101, 102, 103, 104, 105 -- the record is referenced
    DISPLAY my_rec1.*                      -- as a whole

    DISPLAY my_rec1.mem_11 AT 2,2 -- only one record member is referenced

    DISPLAY my_rec1.mem_12 THRU my_rec1.mem_14 AT 3,2
        -- the following members are displayed:
        -- mem_12, mem_13, mem_14

END MAIN
```

You can also assign values to the program record members by returning the values from a function. If a function returns the number of arguments equal to the number of record members, and they are of the same or compatible data types and have the same order, you can use the following syntax:

```
LET record = function()
```

When applied in an application, this may look as it is shown in the example below:

```
DEFINE a_1 RECORD
    var1 VARCHAR(15),
    var2 VARCHAR(15),
    var3 INT
END RECORD

LET a_1 = func()
DISPLAY a_1 -- Displays "Jack Green 32"
END MAIN

FUNCTION func()
DEFINE a,b,c VARCHAR(15)
```



```
LET a = "Jack"
LET b = " Green "
LET c = "32"
RETURN a,b,c
```

You can also pass a record to the function as an actual argument, for example:

```
CALL func(a_1)
```

It is obvious, that the number of the function formal arguments should be the same or more than the number of record members. The record members data type and order should match the arguments data type (or be of compatible data types) and order. If the function has more formal arguments than the record can pass, the arguments get the values starting from the first argument in the arguments list. The "spare" arguments should get values from other actual arguments or from the statements located within the function:

```
DEFINE a_1 RECORD
    var1 VARCHAR(15),
    var2 VARCHAR(15),
    var3 INT
END RECORD

LET a_1 = "Jack", " Green ", "32"

CALL func(a_1)

END MAIN

FUNCTION func(a,b,c,d,e)
    DEFINE a,b,c,d,e VARCHAR(15)
    LET d = "USA"
    LET e = "New Jersey"
    DISPLAY a,b,c
    DISPLAY d, " ", e
END FUNCTION
```

In this example, we have the same program record as in the previous one, but we assign values to it in the MAIN block and then pass these values to the function *func()* as actual arguments. The arguments that don't get values passed from the calling routine get them during the function execution.

A Screen Record

A screen record is a set of fields on a screen form that can be referenced as a single object. You can pass values to a screen record by creating a correspondence between a screen record and a program record.



4GL automatically creates a default screen record which includes form fields that are connected to the same database (or are not connected to any data base). As our forms are formonly, all the fields in them belong to the default formonly record which comprises all the form fields which are not linked to any database. You don't have to specify this default screen record, because it is created automatically.

The members of a screen record are form fields, and like in the program record, they are called *members*.

Screen Record Fields

To create a screen record, you should first add some fields to the SCREEN section of a form and declare them in the [ATTRIBUTES](#) section as described in the previous chapter. All the fields added will belong to the formonly record and you will be able to reference them without having to add anything else to the form file.

All the six form fields (*p_name*, *p_surname*, *p_life*, *deed*, *place*, *p_date*) of the screen form below called "personalities.per" are the members of a default screen record *formonly*. *:

```
DATABASE formonly

SCREEN {
    GREAT PEOPLE

    who:           [p_name]      ]
                    [p_surname]    ]

    was born:      [p_life]

    -----
    what did:      [deed]       ]
    where it happened: [place]   ]
    when:          [p_date]     ]
}

ATTRIBUTES
p_name=formonly.p_name;
p_surname=formonly.p_surname;
p_life=formonly.p_life;
deed=formonly.deed;
place=formonly.place;
p_date=formonly.p_date;

INSTRUCTIONS
DELIMITERS "||"
```

We can create several screen records using these fields. We can divide these fields logically into two groups - those which correspond to personal data will become members of the screen record *personal*, those which correspond to personal achievements will become the members of a screen record called *deeds*. However, all



the fields will still belong to the *formonly* record, because a field can belong to any number of screen records at the same time.

Screen records in the INSTRUCTIONS Section

Programmer-defined screen records are those which need to be declared explicitly, they include only the fields of any types specified in the declaration unlike the default records which include all the fields of a specific type. To declare a programmer-defined screen record, you have to use the INSTRUCTIONS section of the form file. This section is used to declare programmer-defined screen records and screen arrays (we will discuss the latter in the following chapter) as well as to declare the delimiters that indicate the borders of the displayed form fields.

The INSTRUCTION section must be located at the very end of the form file, just after the last item of the ATTRIBUTE clause. The screen record specification must follow the delimiters clause, if any.

The general syntax for the screen record declaration is as follows:

```
SCREEN RECORDS record_name (field1 [, field2, field3...])
```

Here fields are field names declared in the ATTRIBUTES section, not the field tags.

If you add the section below to the form file in the example above (*personalities.per*), you will have three screen records, two of which will be programmer-defined and one will be default (*formonly*):

```
INSTRUCTIONS
  DELIMITERS "<>"
  SCREEN RECORD personal (p_name, p_surname, p_life)
  SCREEN RECORD deeds (deed, place, p_date)
```



Note: The screen record identifier must be unique and can be no longer than 128 bytes. The screen record can be referenced by the program statements only when the screen form including this record is displayed.

You can also use the THROUGH (or THRU) keyword to create a screen array. In this case, the screen array will consist of consecutive fields in the order they are declared in the ATTRIBUTE clause. The following declaration of the *personal* record will have the same effect as in the example above.

```
SCREEN RECORD personal (p_name THRU p_life)
```

Now, if you combine the INSTRUCTIONS part of the form file code with the previous part with the form and fields specification, you will get a form file which contains six fields grouped into three screen records (*formonly*, *personal*, and *deeds*).



Note: You can close the INSTRUCTIONS section with the optional END keyword.

Displaying Values of a Program Record to a Screen Record

There are several ways to display information to a form field. One of them has already been discussed in the previous chapter: it is the DISPLAY TO statement. This statement can also be used to fill a screen record with values of a program record. You can either specify each record member you want to display and then each field you want the values to be displayed to. However, if you want to display all the values of a program record to all the fields of the screen record you can use the following syntax:



```
DISPLAY program_record.* TO screen_record.*
```

where *program_record* stands for a program record name and *screen_record* stands for a screen record name. It is important, that the screen and program records members should correspond in number and data type when you use this statement.

The example below shows how you can display some values of a program record *pers* to *personal* screen record of "personalities.per" form file, if you add the programmer defined screen records to it and compile the form file and source file into one program:

```
MAIN
DEFINE pers RECORD
    f_name, l_name, birth character (20)
END RECORD

LET pers = "Christopher", "Columbus", "1451"

OPEN FORM my_form FROM "personalities"
DISPLAY FORM my_form
DISPLAY pers.* to personal.*
END MAIN
```

As the result, the following will be displayed to the screen (in character and in the GUI modes):

GREAT PEOPLE	
who:	[Christopher]
was born:	[Columbus]

what did:	[]
where it happened:	[]
when:	[]
-	

GREAT PEOPLE	
who:	<input type="text" value="Christopher"/>
was born:	<input type="text" value="Columbus"/>

what did:	<input type="text"/>
where it happened:	<input type="text"/>
when:	<input type="text"/>

You can also enter a value of one of the record members to one screen field. To do this, follow the syntax:

```
DISPLAY record_name.member_name TO record_name.member_name
```

You cannot use the THRU statement to indicate the record members which are to be displayed to the form.



The DISPLAY BY NAME Statement

The BY NAME clause of the DISPLAY statement can be used to display values of the variables to form fields, if the names of the target form fields match the names of the displayed variables. For example, if the source code contains variables named *deed*, *place*, *p_date*, and the form file has fields with the same identifiers, you can use the following code instead of DISPLAY TO:

```
...
DEFINE deed, place, p_date CHAR(30)
LET deed = "discovered America"
LET place = "San-Salvador"
LET p_date = "1492"
DISPLAY BY NAME deed, place, p_date
```

The source code will produce the following result:

```
what did:      [discovered America]
where it happened: [San-Salvador      ]
when:          [1492           ]
```

what did:	discovered America
where it happened:	San-Salvador
when:	1492

You can also use the DISPLAY BY NAME statement to input values from a whole program record to a screen record. If you want to do this, the names of program and screen records need not match, but their members must match in number, order, data type, and names. In this case, you must specify only the program record name after the DISPLAY BY NAME keywords:

```
...
DEFINE
  deeds_rec RECORD deed, place, p_date CHAR(30)
  END RECORD
LET deed_rec = "discovered America", "San-Salvador", "1492"
DISPLAY BY NAME deeds_rec.*
```

You can also use the THRU (THROUGH) keyword in the BY NAME clause, the following example will have the same effect as the two previous ones:

```
DISPLAY BY NAME deeds_rec.deed THRU deeds_rec.p_date
```

The INITIALIZE Statement

The INITIALIZE statement can be used with any variables to assign NULL values to them. There are other opportunities offered by this statement, but they will be discussed later, after the concept of the databases is introduced.

The INITIALIZE statement used to assign NULL values has the following syntax:

```
INITIALIZE variable_list TO NULL
```

The variable list can contain one or more variable of a simple data type, a record or a set of its members, or an array element.



A Programmer Defined Data Type

Q4GL provides you with an ability to create your own data types which can include any of the existing 4GL data types and other user-defined data types. After you declare a new data type, you can use it when declaring program variables. To declare a new data type, you should keep to the following syntax:

```
DEFINE identifier TYPE [UNDER prog_def_type] AS 4gl_data_type
```

Identifier stands for the name of the new data type; *prog_def_type* stands for another user-defined data type that was declared before the current one; *4gl_data_type* stands for any Querix 4GL or user-defined data type.

For example, to define a new data type as a simple data type and use it in your program, you can add such line to your source code:

```
# Declare a new data type named my_dt which is
# a the CHARACTER data type 32 characters long

DEFINE my_dt TYPE AS CHAR (32)

# Then declare variable "a" of my_dt data type
# i.e., a character variable which can contain 32 characters.
# Thus you don't have to declare the size

DEFINE a my_dt
```

After you have specified the length of a new data type (32 in the example above), you **cannot** change it when declaring variables of this data type; for example, you cannot specify the variable *a* as *my_dt(10)*. To define a new data type as a structured data type, you have to specify the elements of this structured data type just as in a usual declaration clause:

```
DEFINE my_dt1 TYPE AS
  RECORD
    var1 VARCHAR(15),
    var2 VARCHAR(15),
    var3 INT
  END RECORD

DEFINE a_1 my_dt1
```

In the example given above, the variable *a_1* is of the *my_dt1* data type, which means, that it is a record consisting of two character members and one integer member. Therefore, you can initialize the variable *a_1* in the following way:

```
LET a_1 = "Jack", "Black", "32"
```

You can modify the previously declared user-defined data type and use it as another new data type. You can add new members to user-defined data types declared as records, but you cannot remove members from them. The UNDER keyword is the tool which is used to perform these modifications.

When you want to modify a previously declared data type, declare a new data type, add the UNDER keyword to the declaration clause, and specify the changes, for example, new record members, as it is shown below:

```
DEFINE my_dt_new TYPE UNDER my_dt AS RECORD

  var4 INT

END RECORD
```



In the example above, *my_dt* is a previously specified data type having three record members. When used after the UNDER keyword, it becomes a basis for the *my_new_dt* user-defined data type, which will have four members: three record members specified for the *my_dt* data type and a new member *var4*. As you can see, the UNDER keyword lets you omit copying the contents of one user-defined data type when you create another one.

Example

The example below manipulates several program and screen records. It illustrates simple and nested program records and the interactions between program and screen records.

```
#####
# This example illustrates the correlations between program and screen records
#####

# First we will declare some customized data types
# which then will be used to declare variables

DEFINE LNAME TYPE AS CHAR(30) -- We define a new simple data type lname;
                                -- the variable of this data type will be
                                -- a character value 30 characters long

DEFINE PERSON TYPE AS RECORD -- we specify a new structured data type PERSON
    first_name,           -- A variable of such data type will be
    last_name  LNAME,     -- a record with five members
    date_birth DATE,      -- of different simple data types,
    age        INTEGER,   -- including a programmer-defined one
    phone      CHAR(14)
END RECORD

DEFINE ADDRES_TYPE TYPE AS RECORD -- another customized RECORD type
    address      VARCHAR(80),
    city         CHAR(20),
    zip_code     CHAR(6),
    country_name CHAR(20)
END RECORD

# Then we enhance the above declared data type
# by adding more members to the record
DEFINE FULL_DATA TYPE UNDER ADDRES_TYPE AS RECORD
    id          INTEGER,
    first_name,
    last_name  LNAME,
    date_birth DATE,
    age        INTEGER,
    phone      CHAR(14),
    units_pd   DECIMAL (5,2),
    price_pu   MONEY(16,2),
    id_card    CHAR(10),
    date_issue,
    date_expire DATE,
    add_data   CHAR(512),
    start_date DATE,
    start_time DATETIME HOUR TO SECOND,
```



```

        end_date      DATE,
        end_time     DATETIME HOUR TO SECOND,
        accept_date   DATE
    END RECORD

MAIN

DEFINE
    first_name,
    last_name   LNAME, -- we specify the last_name variable as the variable
                    -- of the newly declared LNAME data type
    r_person   OF PERSON, -- we specify a variable as a programmer-defined
                    -- structured data type declared above
    r_full_address ADDRES_TYPE, -- OF is optional
    r_all_data  OF FULL_DATA,
    r_full_data_of_person RECORD
        id INTEGER,
        r_person PERSON, -- a nested program record
                           -- in this case a programmer
                           -- defined type saves time
        units_pd    DECIMAL (5,2),
        price_pu    MONEY(16,2),
        id_card     CHAR(10),
        date_issue,
        date_expire DATE
    END RECORD

#####
# The section below is used to assign values to the declared program records
#####

# Below we assign values to simple variables of CHAR data type which we want to
# display to the form fields

LET first_name          = "John"
LET last_name           = "Smith"

#Here we initialize all the elements of all the declared screen records to NULL.
#This is usually done after the records have been used and before using them
#again to remove any values stored in them
INITIALIZE r_person.* ,r_full_address.* ,r_all_data.* ,r_full_data_of_person.* 
TO NULL

# We initialize the members of record r_person
LET r_person.first_name      = "John"
LET r_person.last_name       = "Smith"
LET r_person.date_birth      = "03/08/1965"
LET r_person.age              = YEAR(TODAY) - YEAR(r_person.date_birth)
LET r_person.phone            = "0663850322"

# We initialize the members of record r_full_address
LET r_full_address.address    = "102, Everton Rd."
LET r_full_address.city       = "Liverpool"
LET r_full_address.zip_code   = "123456"
LET r_full_address.country_name = "United Kingdom"

```



```

# We initialize the members of record r_all_data
LET r_all_data.id = 1
LET r_all_data.first_name = "John"
LET r_all_data.last_name = "Smith"
LET r_all_data.date_birth = r_person.date_birth
LET r_all_data.age = YEAR(TODAY) - YEAR(r_all_data.date_birth)
LET r_all_data.phone = "0663850322"
LET r_all_data.units_pd = 5.5
LET r_all_data.price_pu = 9.00
LET r_all_data.id_card = "2397213D"
LET r_all_data.date_issue = "02/01/2000"
LET r_all_data.date_expire = "02/01/2015"
LET r_all_data.address = "102, Everton Rd."
LET r_all_data.city = "Liverpool"
LET r_all_data.zip_code = "123456"
LET r_all_data.country_name = "United Kingdom"
LET r_all_data.add_data =
"Here the additional comments are displayed to a multisegment field. ",
"This field can comprise several lines of text either displayed to it ",
"or entered by a user. In a form file a multi-segment field looks like",
"several fields with the same tag placed one under another. ",
"The WORDWRAP attribute is required to create a multi segment field."
LET r_all_data.start_date = TODAY
LET r_all_data.start_time = CURRENT HOUR TO SECOND
LET r_all_data.end_date = "12/31/9999"
LET r_all_data.end_time = "23:59:59"
LET r_all_data.accept_date = TODAY

# We initialize the members of record r_full_data_of_person
# This includes the initialization of the nested record members.
# Pay attention to the way the values are assigned to the nested record members
# using the structure record.nested_record.member

LET r_full_data_of_person.id = 1
LET r_full_data_of_person.r_person.first_name = "John"
LET r_full_data_of_person.r_person.last_name = "Smith"
LET r_full_data_of_person.r_person.date_birth = r_person.date_birth
LET r_full_data_of_person.r_person.age = YEAR(TODAY) - YEAR(r_person.date_birth)
LET r_full_data_of_person.r_person.phone = "0663850322"
LET r_full_data_of_person.units_pd = 5.5
LET r_full_data_of_person.price_pu = 9.00
LET r_full_data_of_person.id_card = r_all_data.id_card
LET r_full_data_of_person.date_issue = r_all_data.date_issue
LET r_full_data_of_person.date_expire = r_all_data.date_expire

#####
# The section below is used to display values stored in program records to
screen
# fields and records
#####

OPEN WINDOW w_persons AT 1,1 WITH FORM "app_form_1" ATTRIBUTE (FORM LINE 1)

# After we opened the form we display the correct value to the main

```



```
# dynamic label. This label on the form has no colour attribute
# so we specify the colour during the display
# With labels used as headers you don't need to worry about
# the correct coordinates for displaying and the length of the message
DISPLAY "The values of simple variables are displayed to form fields:"
    TO label_main ATTRIBUTE(GREEN, BOLD)

# Displaying simple variables to form fields
DISPLAY first_name, last_name TO s_person.first_name, s_person.last_name
CALL fgl_getkey()
CLEAR FORM

# We change the contents of the label depending on the program execution
DISPLAY "A part of a program record is displayed to form fields:"
    TO label_main ATTRIBUTE(GREEN, BOLD)
# Displaying some of the members of a program record to several fields belonging
# to a screen record
DISPLAY r_person.first_name, r_person.last_name, r_person.date_birth
    TO s_person.first_name, s_person.last_name, s_person.date_birth
CALL fgl_getkey()
CLEAR FORM

DISPLAY "A complete program record is displayed to a screen record:"
    TO label_main ATTRIBUTE(GREEN, BOLD)
# Displaying all the program record members to all the fields of a screen record
DISPLAY r_person.* TO s_person.*
CALL fgl_getkey()
CLEAR FORM

DISPLAY "Simple variables displayed by means of the BY NAME clause:"
    TO label_main ATTRIBUTE(GREEN, BOLD)
# Using the BY NAME clause to display simple variables to form fields. The names
# of variables and fields match.
DISPLAY BY NAME first_name, last_name
CALL fgl_getkey()
CLEAR FORM

DISPLAY "Some program record members are displayed using the BY NAME clause:"
    TO label_main ATTRIBUTE(GREEN, BOLD)
# Displaying some record members to some screen record fields using the BY NAME
# clause. The names of members and fields match.
DISPLAY BY NAME r_person.first_name, r_person.last_name, r_person.date_birth
CALL fgl_getkey()
CLEAR FORM

DISPLAY "All program record members are displayed using the BY NAME clause:"
    TO label_main ATTRIBUTE(GREEN, BOLD)
# Displaying all the record members to all the fields of the screen record using
# the BY NAME clause. The names of record members and the names of the form
# fields match.
DISPLAY BY NAME r_person.*
CALL fgl_getkey()
CLEAR FORM

DISPLAY "Displaying several program records at once with the BY NAME clause:"
```



```

        TO label_main ATTRIBUTE(GREEN, BOLD)
# Displaying two program records to two screen records with the help of the
# BY NAME clause. This is possible only if the names of record members and
# screen fields match.
    DISPLAY BY NAME r_person.* ,r_full_address.*
    CALL fgl_getkey()
    CLEAR FORM

    DISPLAY "Displaying program record r_all_data to the form BY NAME:"
        TO label_main ATTRIBUTE(GREEN, BOLD)
# Displaying the program record values to a formonly record using the BY NAME
# clause. This is possible only if the fields and record members have the same
# names.

# Our form has labels, so if we try to display this record to the form
# using r_all_data TO formonly.* syntax, we will get an error
# for display values for labels are not included into the program
# record. Using the BY NAME clause we successfully display only
# values we need and avoid such an problem.
    DISPLAY BY NAME r_all_data.*
    CALL fgl_getkey()
    CLEAR FORM

    DISPLAY "Displaying a program record with some NULL values:"
        TO label_main ATTRIBUTE(GREEN, BOLD)
# Several record members are set to NULL values using the INITIALIZE statement
# When the record is displayed, the corresponding fields remain empty
    INITIALIZE r_all_data.units_pd THRU r_all_data.add_data TO NULL
    DISPLAY BY NAME r_all_data.*
    CALL fgl_getkey()
    CLEAR FORM

    DISPLAY "Displaying a program record with a nested record:"
        TO label_main ATTRIBUTE(GREEN, BOLD)
# When a program array which contains a nested record is displayed, the members
# of the nested record are also displayed, if they have values.
    DISPLAY BY NAME r_full_data_of_person.*
    CALL fgl_getkey()
    CLEAR SCREEN

END MAIN

```

The Form File

The form file below is called "app_form_1.per" and is used to display the screen records from the source code file above:

```

DATABASE formonly
SCREEN SIZE 26 BY 80
{
```

```

    [label_main
\gp-----[label_1
\g|\g      ID Number:\g[f00      ]
\g|\g      First Name:\g[f01
]-----[image  ]  \|g
]
```



```
\g|\g      Last Name:\g[f02]           ]
\g|\g      Birth Date:\g[f05]     ]\g Age:\g[f6]           ] \g
|\g
\g|\g      Phone:\g[f07]           ]
\g|\g Units per day:\g[f08]     ]\g Price per unit:\g[f09]           ] \g
|\g
\g|-----[label_2]----- |----- | \g
\g|\g ID card:\g[f10]     ]\g Issued on:\g[f11]     ]\g Expires on:\g[f12]   ]
|\g
\g|-----[label_3]----- |----- | \g
\g|\gAddress:\g[f13]           ]\g Zip:\g[f15]   ]\g Country:\g[f17]   ] \g
\g|\g City:\g[f14]           ]\g Zip:\g[f15]   ]\g Country:\g[f17]   ]
]|\g
\g|-----[label_4]----- |----- | \g
\g|[f18]           ]|\g
\g|[f18]           ]|\g
\g|[f18]           ]|\g
\g|[f18]           ]|\g
\g|[f18]           ]|\g
\g|-----[label_5]----- |----- | \g
\g|\gRegistered:\g[f19]   ][f20]     ]\g Dismissed:\g[f21]   ][f22]   ]
|\g
\g|\g Accepted:\g[f23]   ]
\gb-----d\g
}
```

ATTRIBUTES

```
image = formonly.image, config = "default_avatar.jpg", widget = "image";

label_main = formonly.label_main, LEFT, -- will be aligned left
            config = "The main descriptive label", -- will be changed at runtime
            widget = "label"; -- specifies the dynamic label

-- other labels will not be changed at runtime
label_1 = formonly.label_1, CENTER, -- will be centred
          config = "Application form", widget = "label", COLOR = CYAN;
label_2 = formonly.label_2,CENTER, config = "Passport",
          widget = "label", COLOR = CYAN;
label_3 = formonly.label_3,CENTER, config = "Address",
          widget = "label", COLOR = CYAN;
label_4 = formonly.label_4,CENTER, config = "Additional Data",
          widget = "label", COLOR = CYAN;
label_5 = formonly.label_5,CENTER, config = "Registration",
          widget = "label", COLOR = CYAN;

f00 = formonly.id TYPE INTEGER, COLOR = REVERSE;
f01 = formonly.first_name TYPE CHAR;
f02 = formonly.last_name;
f05 = formonly.date_birth TYPE DATE;
f6 = formonly.age TYPE INTEGER;
f07 = formonly.phone;
f08 = formonly.units_pd TYPE DECIMAL;
f09 = formonly.price_pu TYPE DECIMAL,
          COLOR = GREEN WHERE f09 >= 10.00 AND f09 <= 20.00,
          COLOR = YELLOW WHERE f09 > 20.00;
```



```
f10 = formonly.id_card TYPE CHAR;
f11 = formonly.date_issue TYPE DATE;
f12 = formonly.date_expire TYPE DATE;
f13 = formonly.address TYPE VARCHAR;
f14 = formonly.city TYPE CHAR;
f15 = formonly.zip_code TYPE CHAR;
f17 = formonly.country_name;
f18 = formonly.add_data TYPE CHAR,WORDWRAP;
f19 = formonly.start_date TYPE DATE;
f20 = formonly.start_time TYPE DATETIME HOUR TO SECOND;
f21 = formonly.end_date TYPE DATE,COLOR = BLACK WHERE f21 = "12/31/9999";
f22 = formonly.end_time TYPE DATETIME HOUR TO SECOND,
      COLOR = BLACK WHERE f22 = "23:59:59";
f23 = formonly.accept_date TYPE DATE;
```

INSTRUCTIONS

```
DELIMITERS "[ ]"
-- you can use the formonly prefix or omit it
SCREEN RECORD s_person (formonly.first_name, formonly.last_name,
                        date_birth, age, phone)
-- applies to all the fields listed between the specified ones
-- in the ATTRIBUTES section
SCREEN RECORD s_full_address (address THRU country_name)
```

The Image File

This form uses an image widget referencing the image called "default_avatar.jpg". You can use the following image for this purpose. It must be called the same and imported to the source folder and to the location where your form file is placed.





Screen and Program Arrays

An array is the second structured data type in 4GL. It is used to store ordered and named sets of values of the same data type unlike the RECORD data type, which can comprise members of different data types. Whereas program records can contain members of various data types at the same time, a program array can contain elements of only one data type at a time.

The Program Array Declaration

ARRAY variables store one-, two-, and three-dimensional program arrays. The array members, called *elements*, may belong to any data type except ARRAY or DYNAMIC ARRAY.

The syntax of an array declaration is as follows:

```
DEFINE
    array_name ARRAY  [a, b, c] OF Data Type
                            ^_____
                                size
```

The *array name* is an identifier, by which the program references the program array. The requirements for the *array name* are the same as the requirements for other variable identifiers.

The array *size* is an obligatory clause and can be represented by a positive integer that specifies the number of elements within one dimension. The number of array dimensions is determined by the number of the *size* values in brackets. Each dimension can have a different *size*. Each array must have at least one dimension, the rest of the dimensions are optional.

- *a* stands for the number of array elements within the first dimension. It can also stand for the number of elements in a single-dimensional array. For example, *my_arr[5]* will indicate, that the array *my_arr* has only one dimension and consists of five elements.
- *b* stands for the number of array elements within the second dimension. For example, *my_arr[5,10]* indicates that an array *my_arr* is two-dimensional and consists of 5 rows and 10 columns.
- *c* stands for the number of elements within the third dimension of the array. For example, the array *my_arr[5,15,10]* has 5 elements in the first dimension, 15 - in the second, and 10 - in the third.

The minimum value of the *size* identifier is 1, the maximum value is 32,767.

Here is an example of an ARRAY declaration:

```
DEFINE
    my_arr ARRAY [3, 8] of INTEGER
```

Above is a two-dimensional array of INTEGER data type values called *my_arr*, which has 3 lines and 8 columns.



You can declare an array that consists of elements of RECORD data type. If so, you have to specify record members when declaring the program array:

```
DEFINE

    rec_array [100] OF RECORD

        mem_1, mem_2 CHAR(15),

        mem_3 INT

    END RECORD
```

This part of a source code declares a record that has 100 elements of the RECORD data type, each containing two CHARACTER and one INT value. This is the only way to make an array include variables of different data types.

You can also create a record with a member of ARRAY data type:

```
DEFINE my_rec RECORD
    mem1 INT,
    mem2 CHAR(20),
    mem3 ARRAY[100] OF VARCHAR(15)
END RECORD
```

Dynamic Arrays

The DYNAMIC ARRAY data type works and can be referenced the same as the ARRAY data type, but it has some additional features. The difference between the ARRAY and the DYNAMIC ARRAY is that the latter does not have a fixed upper limit for the number of elements and can contain more elements than a static one (a static array can contain 32,767 elements whereas this number for dynamic arrays is 2,147,483,647). If you try to write beyond the bounds of a dynamic array, the program will resize it automatically, whereas you cannot add extra elements to a static program array. The size of the array corresponds to the index of its last element.

The syntax of a single-dimensional dynamic array declaration is as follows:

```
DYNAMIC ARRAY OF 4GL_Data_Type
```

4GL_Data_Type stands for any 4GL or programmer-defined data type.

An array size will be extended, if needed, when new values are added to it.

```
DEFINE darr DYNAMIC ARRAY OF INTEGER
```



```
FOR i = 1 TO 500  
  
    LET darr[i] = i  
  
END FOR
```

The declaration of a multi-dimensional dynamic array is as follows:

```
DYNAMIC ARRAY WITH integer_expression DIMENSIONS OF 4GL_Data_Type
```

Here the integer expression is an expression returning a positive integer from 1 to 3 denoting the number of array dimensions.

Here is an example of a multi-dimensional dynamic array:

```
DEFINE darr1 DYNAMIC ARRAY WITH 2 DIMENSIONS OF INTEGER  
  
FOR i = 1 TO 10  
  
    FOR ii = 1 to 10  
  
        LET darr1[i,ii] = i+ii  
  
    END FOR  
  
END FOR
```

A multi-dimensional array can consist of elements of structured data types, i.e., records and arrays:

```
MAIN  
  
DEFINE  
  
my_arr_2 DYNAMIC ARRAY WITH 2 DIMENSIONS OF  
  
RECORD  
  
    v_int      INTEGER,  
  
    v_char    CHAR(20)  
  
END RECORD,  
  
    i,y      INTEGER  
  
FOR i = 1 TO 2  
  
    FOR y = 1 TO 3  
  
        LET my_arr_2[i,y].v_int  = i*10+y  
  
        LET my_arr_2[i,y].v_char = "element ",i*10+y
```



```
END FOR  
END FOR  
END MAIN
```

If a multi-dimensional dynamic array elements are arrays, these arrays can be of any structure, available for usual arrays.

Here is an example of a two-dimensional dynamic array the elements of which are two-dimensional dynamic arrays:

```
MAIN  
DEFINE  
my_arr_2_2 DYNAMIC ARRAY WITH 2 DIMENSIONS OF  
DYNAMIC ARRAY WITH 2 DIMENSIONS OF INTEGER,  
i1,i2,  
j1,j2  INTEGER  
FOR i1 = 1 TO 2  
FOR i2 = 1 TO 2  
FOR j1 = 1 TO 3  
FOR j2 = 1 TO 3  
LET my_arr_2_2[i1,i2][j1,j2] = i1*1000+i2*100+j1*10+j2  
END FOR  
END FOR  
END FOR  
END FOR  
END MAIN
```

Multi-dimensional arrays cannot be used in the DISPLAY ARRAY statement discussed later in this chapter. The other restrictions on the dynamic array elements are the same as the restrictions on the static array elements described above.

For adding and removing array elements and resizing the array special methods are used. Like it was already mentioned earlier, methods are prefixed. The methods are described at the end of this chapter.



Referencing Array Elements

As with a program record, a program array can be referenced as a whole object, or you can reference each element of a program array individually.

To refer to an array element, you have to indicate its coordinates in square brackets. The general syntax of reference to an array element:

```
array_name[coordinate1, coordinate2, coordinate3]
```

The number of coordinates specified must correspond to the number of the array dimensions. For example,

```
my_arr[3,5]
```

indicates the 5th element of the 3rd row of a two-dimensional array *my_arr*.

Before a program array can be used in the program, its elements must attain some values. There are several ways of assigning values to a program array. In this chapter, we will discuss three of them.

You can assign values to each array element separately, but if your array is quite large, it will take too much of effort and a huge amount of code. The easier way to assign values to array elements is to use conditional loops. This method can be used when the correlation between the element values can be formalized:

```
DEFINE darr ARRAY [50,50] OF INTEGER, -- array declaration
               i,k INTEGER,           -- counters declaration
               val INTEGER            -- variable used for assigning values

LET i=1
    LET k=1
    LET val=1

FOR i=1 TO 50      -- scrolling through the first dimension
    FOR k=1 TO 50 -- scrolling through the second dimension
        LET darr[i,k]=VAL*2
            -- assigning value to the element, which
            -- has coordinates specified by the values
            -- of the counters

        LET VAL=VAL+1 -- changing the value to be assigned
    END FOR
END FOR
```



Another way to assign values is to list the values of the program array elements after the LET statement. It may prove useful when there is no logical connection between the values of the elements. This method is applicable only to one-dimensional arrays. If you apply this method, you must list all the array elements in one LET statement, otherwise an error will occur. The values of the elements should be separated with commas:

```
DEFINE my_arr1 [3] OF INTEGER
LET my_arr1 = 11, 56, 23654
```

If the array has two or three dimensions and you can't input its values by means of a conditional loop, you'll have to assign values separately for each element of your array:

```
My_arr1 ARRAY [2,3] OF CHAR(15)

LET my_arr[1,1] = "One-one "
LET my_arr[1,2] = "One-two "
LET my_arr[1,3] = "One-three"
LET my_arr[2,1] = "Two-one"
LET my_arr[2,2] = "Two-Two"
LET my_arr[2,3] = "Two-three"
```

The disadvantage of this method is that such values assignment needs much space to be written, isn't convenient when is applied to large arrays, and needs much programmer attention in order for him not to miss an element.

On the other hand, this method can be used when the array is already declared and filled with values, and you have to change values of only one or several elements of the array.

If you have an ARRAY of RECORD structure, you should keep to the following scheme of values assignment:

```
LET array_name[coordinate].record_member = value

LET rec_array[1].mem_1 = "Record value"
```

You can pass a whole array or values from some of its elements to a function or from it:

```
MAIN

DEFINE my_arr1 ARRAY [3] OF INT,
      my_arr2 ARRAY [3] OF CHAR(3)

LET myarr_1 = 1,12,234

CALL func1(myarr_1)-- passes values 1, 12, and 234 to the function
```



```
LET myarr_2 = func2()-- gets the values from the function
DISPLAY myarr_2 -- displays "a bc def"
END MAIN

FUNCTION func1(a,b,c)
DEFINE a, b, c VARCHAR(15)
DISPLAY a, " ", b, " ", c -- Displays "1 12 234"
END FUNCTION

FUNCTION func2()
DEFINE arr_f ARRAY[3] OF CHAR(3)
LET arr_f = "a", "bc", "def"
RETURN arr_f --returns "a", "bc", "def" to the program array in the
-- MAIN block
END FUNCTION
```

If the array elements are represented by records, the 4GL passes the values to or from their members in the normal way.

Referencing Elements of a Dynamic Array

Dynamic arrays do not have the upper bond, but their elements can be referenced in the same way as static array elements. However, if you reference an element beyond the current array bond the following result will occur:

- If you reference an element to see or use its value (e.g. in a DISPLAY statement), you will get a runtime error saying that such element does not exist.
- If you reference an array element to assign a value to it, the value will be successfully assigned and the array will be resized. The size of the array will correspond to the index of the referenced element.

```
DEFINE darr DYNAMIC ARRAY OF INTEGER
FOR i = 1 TO 100 -- now the array has 100 elements
    LET darr[i] = i
END FOR
LET darr[120] = 120 -- the array resized to 120 elements
DISPLAY darr[300] -- a runtime error
```



The Screen Array

A *screen array* is a repetitive array that consists of identical rows of form fields. Each row of a screen array comprises a screen record. Each column of a screen array consists of form fields which have the same field tag.

In the SCREEN section of a form file an array usually looks as follows:

```
SCREEN{  
  
    [field1      ]  
    [field1      ]  
    [field1      ]  
  
}
```

This is similar to a multi segment field specification except that a screen array requires a declaration to be made in the INSTRUCTIONS section of a form.

Screen arrays are also declared in the INSTRUCTIONS section of a form file. The syntax of screen array declaration is similar to the syntax of screen record declaration, it is as follows:

```
SCREEN RECORD array_name[size] (elements list)
```

The elements of a screen array must have the form *element_name*, and must be separated with commas. Here is an example of a screen array declaration (the form is called src_arr_form.per):

```
DATABASE formonly  
  
SCREEN {  
  
    Personal  
  
        F_Name      L_Name      AGE  
        [ f_name    | l_name    | age    ]  
        [ f_name    | l_name    | age    ]  
        [ f_name    | l_name    | age    ]  
  
    }  
  
ATTRIBUTES  
    f_name=formonly.f001;  
    l_name=formonly.f002;
```



```
age=formonly.f003;
```

INSTRUCTIONS

```
DELIMITERS " | | "
SCREEN RECORD scr_rec1[3] (formonly.f001, formonly.f002, formonly.f003)
```

As the result, the screen form will contain a screen array consisting of three columns (*f_name*, *l_name*, *age*) and three rows.

Displaying a Program Array to a Screen Array

The DISPLAY ARRAY statement binds screen array fields to the members of a program array and is used to display program array values to field members of the specified screen array. The DISPLAY ARRAY statement can only be used with single-dimensional arrays of RECORD data type. The DISPLAY ARRAY statement syntax is as follows:

```
DISPLAY ARRAY binding_clause [ATTRIBUTE clause] [WITHOUT SCROLL]
[HELP number]
[BEFORE DISPLAY clause]
[BEFORE ROW clause]
[ON KEY clause]
[ON ACTION clause]
[AFTER ROW clause]
[AFTER DISPLAY clause]
}
[END DISPLAY]
```

Input control block

When the DISPLAY ARRAY statement is executed, the program displays the program array to the screen array fields, moves the cursor to the first field of the screen record, and waits until the user presses the ACCEPT key (ESCAPE by default), or one of the scroll keys (by default, they are F3 or PAGEDOWN to scroll forward, and F4 or PAGE UP to scroll backwards). 4GL does not terminate the DISPLAY ARRAY statement until the user presses the Interrupt or the Accept key.

If a character value is larger than the size of the form field, the value will be truncated, if a numeric value is larger than the field, an asterisk (*) appears in such field.

The Binding Clause

The only obligatory clause of the DISPLAY ARRAY statement is the *binding clause*. The binding clause is used to specify the program array to be displayed to the specified screen array. The general syntax of the binding clause is as follows:

```
program_array TO screen_array.*
```



The number of elements in screen and program arrays must match, and the bound elements should support values of the same or of compatible data types. Here is an example of a simplest array displayed

```
DISPLAY ARRAY my_pr_arr TO scr_arr.*
```

You can display only an array of RECORD data type in such a way

The SET_COUNT() Function

Before you can use the DISPLAY ARRAY statement, you need to call a built-in function. This function is used to count the rows of the screen array. It must be called before the DISPLAY ARRAY statement and should specify how many array rows you want to display to the screen array. This function accepts one argument which is the integer expression specifying the number of array rows, e.g.:

```
CALL set_count(100) -- specifies that the array has 100 rows to
display
```

Displaying Array Values

To display values to src_arr_form.per form file from the example above you can use an array of the RECORD data type which contains three members.

You need to declare a single-dimensional array of RECORD data type with size 3 (as we have three rows in the screen array) which contains three members, as shown below.

We use the same form file ("src_arr_form.per") to display this array and the result is the same as in the previous case:

```
MAIN

DEFINE pers ARRAY [3] OF RECORD -- the dimension here means only
                                -- rows
      f_name, l_name CHAR(20),-- the record members match the number
      age INT           -- of columns
END RECORD

# values are assigned to array elements, pay attention to the syntax
LET pers[1].f_name="Mike"
LET pers[1].l_name="McGuire"
LET pers[1].age="34"
LET pers[2].f_name="John"
LET pers[2].l_name="Lock"
LET pers[2].age="53"
LET pers[3].f_name="Kate"
```



```

LET pers[3].l_name="Soyer"
LET pers[3].age="28"

OPEN FORM disp_array FROM "src_arr_form"
DISPLAY FORM disp_array

CALL set_count(3)           -- we have values for three rows and want
                            -- to display them
DISPLAY ARRAY pers TO scr_recl.*
END MAIN

```

The example above will have the following effect:

F_Name	L_Name	AGE
Mike	McGuire	34
John	Lock	53
Kate	Soyer	28

F_name	L_name	AGE
Mike	McGuire	34
John	Lock	53
Kate	Soyer	28

Screen Array Colour Script Options

There is a number of script options that influence the colours of the displayed objects. The effect of these options can be seen only for a limited set of objects and statements. These can be the screen records and the DISPLAY ARRAY statement.

We have earlier discussed such colour options as [background and inactive](#). There are three more colour options that can influence the view of your application.

- **Foreround** option - specifies the foreground colour, or the font colour, for all the displayed objects.
- **BackgroundHighlight** option - specifies the background colour which will be used during arrays input and display to highlight the current row.
- **ForegroundHighlight** option - specifies the font colour for the current row during the an array input or display.

The syntax of these colour options specification is similar to that of other colour options:

```

*background.mygreen: ##98FB98 -- pale green
*foreground.mygreen: #006400
*inactive.mygreen: #006400 -- dark green
*backgroundHighlight.mygreen: #FFFF00 -- yellow
*foregroundHighlight.mygreen: #228B22 -- forest green
*color: mygreen

```



When this colour options are applied to your application, the background colour will be a light green, the colour of the text on this background is dark green. The inactive objects will also be coloured in dark green.

If an array will be displayed to the form, the current line will be highlighted in yellow and the font there will be of a darker green colour.

The Screen Grid

In the examples above, we had to add static labels to the screen form in order to create column headers. There is another way to create headers which will make them dynamic and more convenient to read. You can make column headers out of the first field of each column. To do this, you have to specify a screen grid instead of the screen array in the INSTRUCTIONS section, to specify the fields that are to become headers, and to specify the headers content.

To replace a screen array with a screen grid, specify the GRID keyword instead of the RECORD keyword in the INSTRUCTIONS section. For example,

```
SCREEN RECORD scr_rec1[3]
(formonly.f001, formonly.f002, formonly.f003)
```

Should be replaced by

```
SCREEN GRID scr_rec1[3]
(formonly.f001, formonly.f002, formonly.f003)
```

It is important, that a screen grid does not add a line to the screen array in order to create headers. It takes the first line of the specified array. The column headers will be empty unless you specify them in the ATTRIBUTES section when describing the fields using the following syntax:

```
OPTIONS = "HEADER ='header name' KEY = 'key name'"
```

For example:

```
ATTRIBUTES

f_name=formonly.f001, OPTIONS = "HEADER ='FirstName' KEY = 'F9'";
l_name=formonly.f002, OPTIONS = "HEADER ='LastName' KEY = 'F10'";
age=formonly.f003, OPTIONS = "HEADER ='Age' KEY = 'F11'";
```

The "Key" here specifies what key press the program will indicate when user clicks on the header. The action that the program must perform in this case is specified in the "ON KEY" clauses of output or input statements. The description of action specification is given below in this chapter.

Therefore, when you change a screen array to a screen grid and delete the static labels used as column names from the form, you will get the following view of your form:



PERSONAL		
FirstName	LastName	Age
Mike	McGuire	34
John	Lock	53

Screen grids change the view only in the GUI mode. In character mode, the screen grid looks the same as the rest of the screen array.

You can change the grid headings at runtime. To do it, use the *fgl_grid_header()* function. The function affects the grid headers only in the current form. You can also use the function to change the alignment of the header text and to change the key or event associated with the mouse click on the particular header.

The syntax of the function invocation is:

```
CALL fgl_grid_header ("grid_name", "field_name", "label", "allignment", "key")
```

The alignment and key parameters are optional. Using this function you assign new parameters to the specified grid header.

You can display different program arrays to one and the same screen array - you can change the headers depending on the information displayed. For example, after we have displayed the personal data, we can use the same form to display information about companies:

```
DISPLAY ARRAY pers TO scr_recl.* --displays a screen record to the screen
                                         -- grid with headers specified in the form file
# Changes greed headers of the three columns:
CALL fgl_grid_header ("scr_recl[1]", "f_name", "Company", "Right", "end" )
CALL fgl_grid_header ("scr_recl[1]", "l_name", "City", "Right", "end" )
CALL fgl_grid_header ("scr_recl[1]", "age", "Code", "Right", "end" )

# Displays another program array to the same screen grid
DISPLAY ARRAY pers2 TO scr_recl.*
```

After the last DISPLAY ARRAY statement is executed, the user will see the following display result:



PERSONAL		
Company	City	Code
ASDF	New York	1123
Anna-B	London	43343

Optional Clauses

The DISPLAY ARRAY can include a number of optional clauses. These are the ATTRIBUTE clause and the ON KEY clause which are described below. It can also contain a HELP clause to specify the help message to be displayed, if a user presses the help key during the displaying. The syntax and restrictions of the HELP clause as well as the help key are typical and are described for the [PROMPT](#) statement.

The ATTRIBUTE Clause

The ATTRIBUTE clause of the DISPLAY ARRAY statement allows you to change the appearance of the displayed result. The attributes specified in this clause (except for the CURRENT ROW attribute that is described below) are applied to all fields of the target screen array.

The DISPLAY ARRAY attributes are mostly the same as those that we have discussed before for the DISPLAY statement (NORMAL, BOLD, RED, REVERSE, etc.):

```
DISPLAY ARRAY pers TO scr_recl.* ATTRIBUTE (RED, REVERSE)
```

This line will make the values be displayed in black font on the red background.

The DISPLAY ARRAY statement can have an attribute which cannot be applied to any other statements described so far in this manual; it is the CURRENT ROW DISPLAY attribute. Its syntax is:

```
ATTRIBUTE (CURRENT ROW DISPLAY = "attribute1 [, attribute2...]" )
```

The CURRENT ROW DISPLAY attribute is used to highlight the current row of the screen array (where the cursor is located). It specifies attributes that will be applied to it and that will override the general attributes.

If the screen array contains of only one row, the CURRENT ROW DISPLAY attributes will be applied to this row. Here is an example of this attribute usage:

```
DISPLAY ARRAY pr_arr TO scr_arr.*  
ATTRIBUTE (GREEN, CURRENT ROW DISPLAY = "GREEN, REVERSE" )
```

The example above will cause the screen array values to be displayed with green font, and the current row will have the reversed video:



Personal Data		
F_Name	L_Name	AGE
Mike	McGuire	34
John	Lock	53
Kate	Soyer	28

Personal data		
Mike	McGuire	34
John	Lock	53
Kate	Soyer	28

Scrolling During the DISPLAY ARRAY Statement

4GL provides several keys that can be used for scrolling through a screen array:

- DOWN and RIGTH arrow keys move the cursor down one row when pressed. If the cursor is located at the last row, it will be moved up one row. If the last rows of program array and screen array match, 4GL displays a message warning the user that there are no more rows below.
- UP and LEFT arrow keys move the cursor up one row when pressed. If the cursor is already located at the first row, the keys will move it down one row. If the first rows of program and screen arrays match, 4GL displays a message warning the user that there are no more rows above.
- F3 key is used to scroll the display to the next full page of program array records.
- F4 key is used to scroll the display to the previous full page of program array records.

The F3 and F4 keys can be replaced by other keys by means of the NEXT KEY and PREVIOUS KEY options of the OPTIONS statement, the keys which can be used here are the keys typically used on the OK KEY clauses of the statements and in other OPTIONS key declarations. E.g.:

```
OPTIONS PREVIOUS KEY "CONTROL-P"
OPTIONS NEXT KEY "CONTROL-N"
```

The WITHOUT SCROLL Clause

The WITHOUT SCROLL clause consists only of the WITHOUT SCROLL keywords. If the WITHOUT SCROLL keywords are specified in the DISPLAY ARRAY statement, the statement is executed the following way:

- The program displays the program array to the screen array;
- The program control is passed to the statement which follows the DISPLAY ARRAY statement without waiting for Accept key to be pressed by the user;
- All scroll keys are disabled;
- The program ignores the HELP key and all the optional clauses except the ATTRIBUTE clause.

If you use some optional clauses together with the WITHOUT SCROLL clause, they won't be executed, because the program won't wait for the Accept key to be pressed and leave the DISPLAY ARRAY statement just after it encounters the WITHOUT SCROLL keywords.

In the following example, the green colour and bold font will be applied to the displayed array, but the help message won't be available:

```
DISPLAY ARRAY pr_arr TO scr_arr.*
```



```
ATTRIBUTE (GREEN, BOLD)
HELP 102
WITHOUT SCROLL
PROMPT "Enter the date" FOR dat
```

After the array is displayed, a prompt message appears, but the displayed array will stay in the screen unless this statement displays something else to the form fields or moves to another 4GL window. Although you cannot use the scroll keys to scroll the array displayed with the WITHOUT SCROLL clause, in the GUI mode, you can use the mouse cursor to do it.

The BEFORE DISPLAY Clause

The BEFORE DISPLAY clause is used to specify the actions that the program must perform before any values are displayed to the screen array. The BEFORE DISPLAY clause is optional but it must be unique within one DISPLAY ARRAY statement.

The following BEFORE DISPLAY clause prompts for the user name before the display:

```
BEFORE DISPLAY
  PROMPT "Enter your name: " FOR us_name
END DISPLAY
```

The EXIT DISPLAY Statement

The EXIT DISPLAY statement is used to terminate the DISPLAY ARRAY. When the EXIT DISPLAY statement is executed, the program passes control to the first statement that follows the END DISPLAY keywords and ignores all the statements that are located between the EXIT DISPLAY and END DISPLAY keywords. These keywords can be used in a BEFORE DISPLAY clause or in any other clause of the control block.

For example, this part of the source code will terminate the array display if the user name does not match the one stored in the program memory:

```
BEFORE DISPLAY
  PROMPT "Enter your name: " FOR us_name
  IF us_name NOT MATCHES permitted_n
    THEN EXIT PROMPT
  END IF
END DISPLAY
```



The BEFORE ROW Clause

The BEFORE ROW clause allows you to specify some actions that are to be performed before the cursor moves to another row. When we say "a row", we mean a line within a screen array. The DISPLAY ARRAY statement can contain only one BEFORE ROW block.

In the following example, when the cursor moves to the next line of the screen array, the value of the variable *i* increases by one:

```
BEFORE ROW  
LET i = i+1  
END DISPLAY
```

The ON KEY Clause

As with the PROMPT statement, the ON KEY block of the DISPLAY ARRAY statement is used to specify a set of statements that are to be executed when the user presses one of the keys specified in parentheses that follow the ON KEY keywords.

If you use the ON KEY clause to specify several keys for one statement block, you should separate the key names with commas. The general syntax of the ON KEY clause is:

```
ON KEY (key name1 [, key name 2, key name 3...]) statement_list
```

The keys that can be specified in the ON KEY clause of the DISPLAY ARRAY statement and the restrictions and rules you have to follow when you specify a key, are similar to the keys and rules which are in effect for the [ON KEY clause](#) of the PROMPT statement.

When the user presses one of the keys specified in the ON KEY block, the program executes the statements of this block and then re-activates the form, so that the user can continue working with fields. After the statements in the ON KEY block are executed, the cursor is positioned to the same field as before the ON KEY block execution, unless the EXIT DISPLAY statement has been executed.

If you use one or several ON KEY clauses in one DISPLAY ARRAY statement, you have to add the END DISPLAY keywords after the last ON KEY clause. Otherwise the END DISPLAY keywords are not required.

The following ON KEY clause displays the current date and time when CONTROL-t keys are pressed:

```
DISPLAY ARRAY pr_arr TO scr_arr.*  
ON KEY (control-t) DISPLAY TODAY, " ", CURRENT  
END DISPLAY
```

Like the INPUT, the INPUT ARRAY statement has an optional `infield()` operator in the ON KEY clause, which allows the programmer to trigger field-specific actions. This extension is applicable only for Lycia II and above. Its syntax is as follows:

```
ON KEY(key1 [,key2...]) INFIELD(field1 [,field2...])
```



The ON KEY clause with the infield() operator will be executed, only if the cursor is located in one of the fields specified in the infield() operator, when the referenced key is pressed. In other cases such ON KEY clause will be ignored. If there is another ON KEY clause referencing the same key which has no infield() operator, it will be executed then the cursor is outside of the fields specified in the infield() operator, like in the example below:

```
INPUT ARRAY my_arr FROM scr_arr.*  
ON KEY(F5)  
CALL fgl_messagw_box("Global ON KEY event")  
ON KEY(F5) INFIELD(F2,F3)  
CALL fgl_messagw_box("ON KEY event for fields f2 and f3")  
END INPUT
```

The AFTER ROW Clause

You must specify the AFTER ROW clause in case you want the program to perform some actions after the cursor leaves a row. There can be only one AFTER ROW clause in the DISPLAY ARRAY statement.

In the following example, the AFTER ROW clause displays the message and then erases it:

```
DISPLAY ARRAY pers TO scr_recl.*  
AFTER ROW  
DISPLAY "You've left the row" AT 2,2  
SLEEP 1  
DISPLAY " " AT 2,2  
END DISPLAY
```

The AFTER DISPLAY Clause

The AFTER DISPLAY clause specifies the actions that the program must take after the display is performed but before the DISPLAY ARRAY statement is terminated and the program starts executing the statements following the END DISPLAY keywords. The DISPLAY ARRAY statement can contain only one AFTER DISPLAY statement.

The program executes the AFTER DISPLAY statement, if:

- The user presses the Accept key
- The user presses the Interrupt key and no DEFER INTERRUPT statement is in effect
- The user presses the Quit key and no DEFER QUIT statement is in effect

The program does not execute the AFTER DISPLAY statement, if:



- The user presses the Quit or Interrupt keys when the corresponding DEFER statement is in effect
- The program executes the EXIT DISPLAY statement within one of the optional clauses.

Below is an example of an AFTER DISPLAY clause application:

```
DISPLAY ARRAY pers TO scr_rec1.*  
  
AFTER DISPLAY  
  
    DISPLAY "The display is over" AT 2,2  
  
    SLEEP 2  
  
END DISPLAY
```

The CONTINUE DISPLAY Keywords

The CONTINUE DISPLAY keywords are used to return the program control from a control block to the DISPLAY ARRAY statement. When the program encounters the CONTINUE DISPLAY keywords, it skips all the subsequent statements in the current control block and returns the cursor to the first field of the first row of the screen array. The array is redisplayed. However, if the statements in one of the previous control blocks have displayed the values that are not included to the array to one or several fields of the screen array, these values won't be replaced by the initial ones.

These keywords may be useful when you need to return the program control from a deeply nested conditional loop and continue displaying the array.

The END DISPLAY Keywords

The END DISPLAY keywords are used to terminate the DISPLAY ARRAY statement. These keywords are optional. They are required only when the following conditions are met:

- There is one or more ON KEY blocks in the DISPLAY ARRAY statement
- The DISPLAY ARRAY statement is specified as a part of the ON KEY block of the PROMPT statement or of another DISPLAY ARRAY statement.

The Button Widget

The button field is used to create a button which the user can click. When the user clicks a button, it sends a keypress or an action to the 4GL application. The keypress acts the same as if the user had pressed the key directly on the keyboard.

When you use the DISPLAY TO statement to send some text to a button field, this text becomes a new button label. By default, the buttons are disabled. Therefore, if you want to allow the user to click a button, you have to enable it by the DISPLAY "!" TO *button* statement. To disable a key, use the DISPLAY "*" TO *button* structure.

When you run the application in the character mode, the buttons are displayed as usual fields and cannot be clicked.



The button specification in the ATTRIBUTES section should correspond to the following syntax:

```
Field_tag = database.field_name  
config = "key_or_action {label}"  
widget = "button"
```

For example, let's add a field tagged *my_but* to the form with the screen array. To make a button of this field, specify its attributes as follows:

```
# the form file name is "My_form"  
DATABASE formonly  
  
SCREEN {  
  
    Personal data  
  
    [ f_name | l_name | age ]  
    [ f_name | l_name | age ]  
    [ f_name | l_name | age ]  
  
    [ my_but ]  
}  
  
ATTRIBUTES  
  
...  
...  
  
my_but = formonly.mbutton,  
        config = "F1 {My button}",  
        widget = "button";  
  
INSTRUCTIONS  
DELIMITERS "[ ]"
```

If you display this form with a usual DISPLAY FORM or OPEN WINDOW WITH FORM statement, and specify no appropriate ON KEY clause in the DISPLAY ARRAY statement, the button, when clicked, will cause no program actions.

You have to make two steps to activate the button and make it influence the program workflow. First, you have to activate the button with the DISPLAY "!" TO *button* statement. Second, you have to specify some program actions that are to be performed when the key specified in the field attributes is pressed.

For example, you can add the ON KEY clause to the DISPLAY ARRAY statement:

```
OPEN FORM f1 FROM "my_form"
```



```
DISPLAY FORM f1

DISPLAY "!" TO mbutton

DISPLAY ARRAY pers TO scr_recl.*

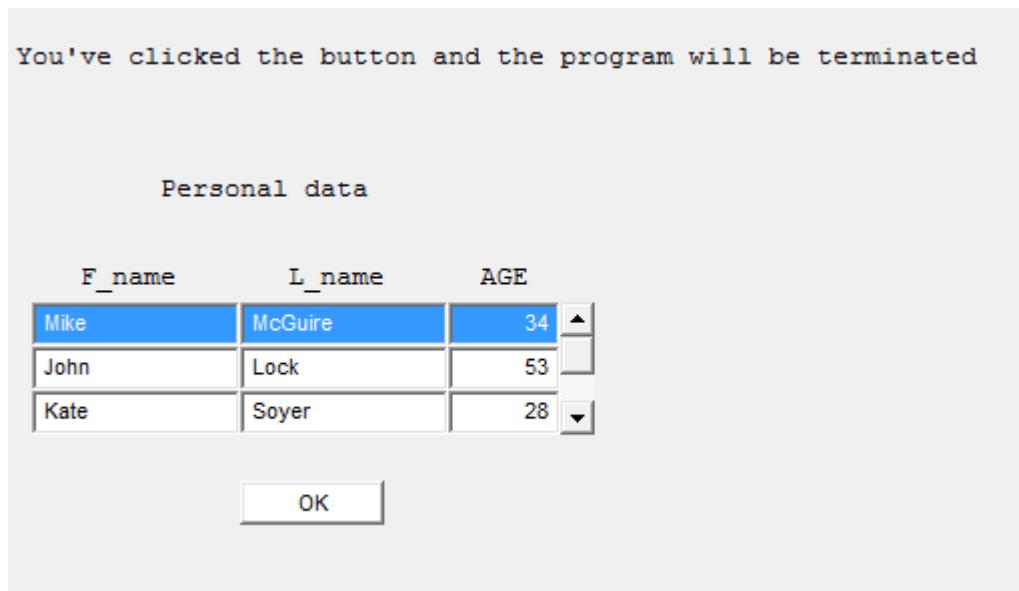
ON KEY (F1) display "You've clicked the button and the program
will be terminated" at 2,2

SLEEP 2

EXIT PROGRAM

END DISPLAY
```

Now, when the button is specified and all the necessary commands are added to the source code, our form has the "OK" button which displays a message to the screen and terminates the program execution. When the application is run and the user clicks the button, they see the following before the program is closed:



The ON ACTION Clause

You can specify not only keys, but also actions for BUTTON FIELD. The main difference between the action and the key is that you can specify any name for the action, and this name isn't bound to reserved keys.

The actions assigned to the buttons can be used in ON ACTION clauses. The ON ACTION clause is very similar to the ON KEY clause. It contains statements that the program executes when the user presses a button or another widget on the form for which the corresponding action is specified.

Unlike the ON KEY clause, which does not necessarily refers to a form element, the ON ACTION clause can be activated only from a widget.



The action (also called the event) is specified after the *config* keyword in the field attributes. The same action name is to be specified within the ON ACTION clause. The general syntax of an action specification is:

```
config = "action {Button_name}" -- in form file
```

and

```
ON ACTION ("action", "alternative_event") -- in the .4gl file
```

The DISPLAY ARRAY statement can contain any number of ON ACTION clauses, each having up to four actions enclosed in quotation marks and separated by commas.

The ON ACTION clause can contain any 4GL statements. These statements are executed when the user clicks the widget to which the action is assigned. When the program starts executing the ON ACTION statements, it deactivates the current form and reactivates it when the ON ACTION clause is fully executed, unless it has the EXIT DISPLAY statement. If there is an EXIT DISPLAY statement in the ON ACTION clause, the program will be terminated. The example below represents the action declaration in the form file and activation in the ON ACTION clause:

```
config = "action {Button_name}" -- in form file
...
ON ACTION ("action", "alternative_event") -- in the .4GL file
```

In a program, this scheme can be realized as follows:

- In a form file

```
my_but = formonly.mbutton,
config = "end Close",-- the action name is "end"
widget = "button";
```

- In a 4GL file

```
DISPLAY ARRAY pers TO scr_recl.*
ON ACTION ("end")
EXIT PROGRAM
END DISPLAY
```

When the application is run and the form is displayed, the user can click the *Close* button to exit the program.



Note: If a keypress or action associated with a button cannot be recognized by the program, it is possible that the key name will be displayed to the current input field when the button is pressed.

The ON ACTION clause of the INPUT and INPUT ARRAY statement can have the optional operator *infield()* which functions in the same way as the *infield()* operator for the ON KEY clause described above in this chapter. It has the following syntax:



```
ON ACTION(action1 [,action2...]) INFIELD(field1 [,field2...])
```

Finding the Size of a Screen and a Program Array at Runtime

Sometimes, it is useful to refer to the sizes of screen and program arrays, not to the actual data. Therefore, Q4GL supports two functions to do this. They are *fgl_scr_size()* and *sizeof()* functions. They can be useful in evaluating the volume of data that are to be inputted, or to check whether the stored data can fit the current screen array, etc.

The *fgl_scr_size()* function is used to return the declared size of the specified screen array at runtime. The function accepts one argument which is the name of a screen array in the currently opened form and returns an integer number which identifies the number of screen records in that screen array.

If you specify the array name directly, into quotes, and by means of a variable containing such name, for example:

```
LET scrname = "screc1"  
LET a = fgl_scr_size(scrname)  
LET a = fgl_scr_size("screc1")
```

The last two lines will have the same effect when the program processes them.

The *sizeof()* function is used to return the number of elements allowed in the array or the maximum length of a character variable. The argument of the function is the name of the variable or the array the size of which we want to return.

For example, we have an array of CHAR variables, and we want to return the size of the array:

```
DEFINE marr ARRAY[10] OF CHAR(20)  
DISPLAY "The array[10] size is ", sizeof(marr) at 2,2
```

In this case, the *sizeof()* function returns 10. If you have a multi-dimensional array, the *sizeof()* function will return only one of them. The number of the dimension which size you want the function to return is to be specified in square brackets following the array identifier. If you do not specify the number, the function will return the number of elements in the first dimension:

```
DEFINE marr ARRAY[10,15,30] OF CHAR(20)  
  
DISPLAY "1-st dimension is ", sizeof(marr) at 2,2 -- Returns 10  
  
DISPLAY "2-nd dimension is ", sizeof(marr[2]) at 2,4 -- returns 15  
  
DISPLAY "3-d dimension is ", sizeof(marr[3]) at 2,6 -- returns 30
```

You can return the length of elements of the ARRAY OF CHAR. To do it, specify the number of the element after the dimensions specification:

```
DEFINE marr ARRAY[10,15] OF CHAR(20)  
  
DISPLAY "Element length is ", sizeof(marr[1,2]) at 2,8
```



In this example, the `sizeof()` function will return the size of the second element of the first line, which is 20.

Managing Dynamic Arrays

To manage dynamic arrays special methods are used:

- `Append(value)` - adds a new element to the end of a dynamic array with the specified value.
- `Clear()` - deletes all the elements from the array and sets its size to 0.
- `Delete(first [,last])` - deletes the element with the specified index or a number of elements between the specified range of indexes.
- `GetSize()` - returns the size of a single-dimensional dynamic array.
- `Insert(index [,value])` - inserts an element to the specified place in the array; if the value is specified, inserts the element containing the value, otherwise the element contains the default value.
- `Resize(size)` -resizes the array to the given size.

```
DEFINE darr ARRAY OF CHAR(20)

...
FOR i = 1 to 10
    LET darr[i]=i
END FOR

DISPLAY darr.getsize() -- returns 10
CALL darr.resize(20) -- resizes array to 20 elements 10 of which contain 0
CALL darr.delete(11, 20) -- deletes the last 10 elements (array size = 10)
CALL darr.append(6) -- appends an element with value 6 (array size = 11)
CALL darr.insert(5,100) -- inserts an element with value 100 at the 5th
                        -- position (array size = 12)
CALL darr.clear() -- sets the array size to 0
```

Retrieving Information about the Current Window

Q4GL supports a function which returns information about the parameters of the currently active application window. This is the `fgl_window_getoption()` function.

The function syntax is:

Fgl_window_getoption(option)

The function can have only one option as an argument. All the possible options and their descriptions are described in the table below:

OPTION	DESCRIPTION
x	The column position
y	The line position
Width	The width
Height	The height



Border	TRUE if window has a border
Formline	The form line
Menuline	The menu line
Commentline	The comment line
Messageline	The message line
Errorline	The error line
Insertkey	The value of <i>insertkey</i>
Deletekey	The value of <i>deletekey</i>
Nextkey	The value of <i>nextkey</i>
Previouskey	The value of <i>previouskey</i>
Acceptkey	The value of <i>acceptkey</i>
Helpkey	The value of <i>helpkey</i>
Abortkey	The value of <i>abortkey</i>
Inputwrap	TRUE if input wrap option is used
Fieldorder	TRUE if fieldorder is used

The function returns the same values for the keys as the *fgl_getkey()* function.

The simplest sample of the *fgl_window_getoption()* function is:

```
OPEN WINDOW w1 AT 2,2 WITH 15 ROWS, 30 COLUMNS
ATTRIBUTES (BORDER)

DISPLAY "column ", fgl_window_getoption("x") AT 2,2 -- returns 2
DISPLAY "raw      ", fgl_window_getoption("y") AT 3,2 -- returns 2
DISPLAY "Border   ", fgl_window_getoption("border") at 4,2 -- returns 1
```

Example

The example below is designed in a form of a ring menu. You can view how one- and multidimensional arrays are handled as well as the arrays of records. It also illustrates the notion of the screen grid.

```
#####
# This example illustrates the correlations between program and screen arrays
#####

DEFINE

#We declare the program array of records as a module variable
# so that init_a_country_list()function could use it
# This array will be used with the DISPLAY ARRAY statement

a_country_list DYNAMIC ARRAY OF RECORD          -- We specify this array as a
                                                -- dynamic one, because we
country_id    CHAR(3),   -- don't know how much elements
code_lit      CHAR(3),   -- the array will contain
country_name  CHAR(50)   -- and we don't want it to
                           -- take extra disk space
END RECORD,                                     -- variable to count the number of cursor movements
                                                -- variable to count the number of the rows that
                                                -- the cursor has visited
                                                -- variable to contain the number of the row of a
```



```
-- program array where the cursor is located
num_scr_row,          -- variable to contain the number of the row of a
                      -- screen array where the cursor is located
cnt_arr_elm,          -- variable to the number of elements
                      -- contained in the program array
scr_size,              -- variable to contain the info about the
                      -- declared size of the screen array
size_of INTEGER,       -- variable to contain the data about
                      -- the allowed number of a program array elements
row_info VARCHAR(500), -- variable to contain all the information about
                      -- the selected row
arr_info VARCHAR (300),--variable to contain the information about the
                      -- program and the screen array
v_char     VARCHAR(10),
answer    CHAR,
arr_val   RECORD      -- A record that will contain values from
                      -- country_id  CHAR(3),      -- the current row
                      -- code_lit   CHAR(3),
                      -- country_name CHAR(50)
                      END RECORD,
head_id   STRING,
head_text STRING

MAIN

# We check whether the application is run in GUI mode
# because screen grid and buttons are
# only available in GUI mode
IF fgl_fglgui()= 0 THEN
  CALL fgl_winmessage("Wrong mode",
                      "Run this application in the GUI mode.",
                      "info")
  EXIT PROGRAM
END IF

# First we initialize the program array
CALL init_a_country_list()

# Then use the menu to show different way to display this array
# and other arrays
MENU "Arrays"
  COMMAND KEY (M)
    "Multi-Dimensional",
    "Displays values assigned to multi-dimensional arrays"
    CALL multi_dem_arr()

  COMMAND KEY (M)
    "Dynamic Arrays",
    "Illustrates how dynamic arrays work"
    CALL dynamic_arr()

  COMMAND KEY (R)
    "Arrays of Records",
    "Displays program records to screen records"
```



```
MENU "Arrays of Records"
    COMMAND "Without Scroll",
    "Displays a program array to a screen array using the WITOUT SCROLL"
        CALL without_scroll()
    COMMAND "With Scroll",
    "Displays a program array to a screen array with scrolling"
        CALL with_scroll()
    COMMAND "Return",
    "Returns to the previous menu"
    EXIT MENU
END MENU

COMMAND KEY (G)
    "Screen Grids",
    "A program record is displayed to the screen greed"
    CLEAR SCREEN
    CALL scr_grid()

COMMAND KEY (Q)
    "Exit",
    "Exit from the program"
    EXIT PROGRAM

END MENU

END MAIN

#####
# The next function of the application deals with screen arrays and the
# DISPLAY ARRAY statement
#####
FUNCTION without_scroll()

# This form contains the screen array
OPEN WINDOW without_scroll AT 5,10 WITH FORM "countr_lst"
ATTRIBUTE (FORM LINE FIRST)

    CALL init_a_country_list() -- this function assigns values to 50
                               -- elements of a_country_list

    CALL set_count(50)       -- as we have 50 elements with assigned
                               -- values, we set 50 as the total array
                               -- row count

# Here we display the program array to the current array
# The first DISPLAY ARRAY statement includes the WITHOUT SCROLL clause
DISPLAY "WITHOUT SCROLL keywords used. Press any key to continue."
    TO header ATTRIBUTE (GREEN, BOLD)

DISPLAY ARRAY a_country_list TO s_country_list.* WITHOUT SCROLL

# The following control block won't be executed, because it follows the
# WITHOUT SCROLL keywords and no control block is allowed,
# if the array is displayed without scroll
BEFORE DISPLAY
```



```
DISPLAY "The DISPLAY statement after the WITHOUT SCROLL keywords" AT 1,1
END DISPLAY

OPTIONS PROMPT LINE LAST - 1

LET answer = "" PROMPT "Press any key to see a screen array" FOR CHAR answer
ATTRIBUTE (YELLOW, BOLD)

CLEAR SCREEN
CLOSE WINDOW without_scroll
END FUNCTION

#####
# This function will display the program array so that the user can scroll
# through it and select some values
#####
FUNCTION with_scroll()

OPEN WINDOW with_scroll AT 5,10 WITH FORM "countr_lst"
ATTRIBUTE (FORM LINE FIRST)

DISPLAY "!" TO accpt      -- Display the "!" sign to the button widget to
DISPLAY "!" TO info       -- activate it
DISPLAY "!" TO cancl

CALL set_count(50)         -- as we have 50 elements with assigned
                           -- values, we set 50 as the total array
                           -- row count

LET num_curs = 0           -- we initialize the counter of the rows visited
LET num_curs_mov = 0        -- we initialize the counter of the cursor movements

DISPLAY ARRAY a_country_list TO s_country_list.* 
ATTRIBUTE (CURRENT ROW DISPLAY = "REVERSE") -- the row with the cursor
                                              -- will have reversed graphics
BEFORE DISPLAY
  CALL fgl_winmessage ("", "The WITHOUT SCROLL keywords\n are not used",
                       "Caution")
  DISPLAY "WITHOUT SCROLL clause is absent" TO headr
  ATTRIBUTE (GREEN, BOLD)

BEFORE ROW -- The BEFORE ROW block counts the number of rows visited
            -- it is executed before the cursor is moved to a new row

  LET num_curs = num_curs+1

AFTER ROW -- The BEFORE ROW block counts the number of cursor movements
            -- it is executed before the cursor is moved from the current row

  LET num_curs_mov = num_curs_mov+1

ON ACTION ("OK")           -- When the user presses the button labelled
                           -- as "OK" the row number of the current row
  LET num_arr_row = arr_curr() -- is assigned to the num_arr_row variable
```



```
EXIT DISPLAY                                -- and finish the display

ON KEY(F1)

# We assign the values of the current array and screen rows to the
# variables to display them in the information message.

LET num_arr_row = arr_curr()
LET num_scr_row = scr_line()
LET cnt_arr_elm = arr_count()

# We make the values above parts of a character value in order to display it to
# the message box
# The first three lines of the value assignment comprise the first line of the
# message that will contain the data from the selected row
# The last three lines of the value assignment make three rows of the message
# which contain information about the screen and the program arrays.

LET row_info =
10 space,a_country_list[num_arr_row].country_id, " ",
a_country_list[num_arr_row].code_lit, " ",
a_country_list[num_arr_row].country_name,
"\n \n - The number of cursor movements is:      " || num_curs_mov,
"\n \n - The number of the rows visited is:      " || num_curs,
"\n \n - The current row of the program array is: " || num_arr_row,
"\n \n - The current row of the screen array is:  " || num_scr_row,
"\n \n - The total number of the records is:      " || cnt_arr_elm

# Now, we can call a message box and use the row_info variable to
# pass the message to it.
CALL fgl_winmessage ("Info", row_info, "info")

ON KEY (F10)           -- here, we specify the actions which are
LET num_arr_row = 0 -- to be taken when the user presses the F10 key
EXIT DISPLAY

END DISPLAY    -- we need these keywords, because we have the ON KEY
                -- clauses; they terminate the DISPLAY ARRAY statement

IF num_arr_row <> 0 THEN
    LET row_info = "You've selected:    ",
        a_country_list[num_arr_row].country_id, " ",
        a_country_list[num_arr_row].code_lit, " ",
        a_country_list[num_arr_row].country_name
    CALL fgl_winmessage ("Info", row_info, "info")
END IF
CLOSE WINDOW with_scroll
CLEAR SCREEN
END FUNCTION

#####
# This function of the program demonstrates how the screen grid works
#####
FUNCTION scr_grid()
```



```
OPEN WINDOW w_countr_grid AT 5,10 WITH FORM "scr_grid"
ATTRIBUTE (FORM LINE FIRST)      -- this form contains the screen array

DISPLAY "!" TO head
DISPLAY "!" TO info
DISPLAY "!" TO acc

DISPLAY "The program array is displayed to the screen grid" TO headr
ATTRIBUTE (GREEN, BOLD)
CALL set_count(50)           -- as we have 50 elements with assigned
                             -- values, we set 50 as the total array
                             -- row count

# Here we display the program array to the current screen grid
DISPLAY ARRAY a_country_list TO s_country_grid.* 
    ATTRIBUTE (CURRENT ROW DISPLAY = "REVERSE")

# The actions for all the keys are listed below, but the action
# for the "Accept" key associated with the button is not,
# because it executes its default function - finishes the display
# even if it is absent from the OK key clauses
ON KEY (F9)
    #The user selects the header to be changed and enters the new value
    LET head_id = fgl_winbutton("Changing a Header",
        "Select the header you want to change", "First Column",
        "First Column|Second Column|Third Column|Cancel",
        "question", 1)
    CASE head_id
        WHEN "First Column"
            # This function changes the specified grid header
            LET head_text = fgl_winprompt(8,10, "Enter the new header name",
                " new name", 15, 0)
            CALL fgl_grid_header ("s_country_grid", "country_id",
                head_text, "Left")
        WHEN "Second Column"
            LET head_text = fgl_winprompt(8,10, "Enter the new header name",
                " new name", 15, 0)
            CALL fgl_grid_header ("s_country_grid", "code_lit",
                head_text, "Left")
        WHEN "Third Column"
            LET head_text = fgl_winprompt(8,10, "Enter the new header name",
                " new name", 15, 0)
            CALL fgl_grid_header ("s_country_grid", "country_name",
                head_text, "Left")
        OTHERWISE
            CALL fgl_winmessage("Cancelled", "Header wasn't changed", "info")
    END CASE

# These keys (F10-F12) are associated with the headers
# and can also be used to change the titles

ON KEY (F10)
    LET head_text = fgl_winprompt(8,10, "Enter the new header name",
        " new name", 15, 0)
```



```

CALL fgl_grid_header ("s_country_grid", "country_id", head_text, "Left")

ON KEY (F11)
    LET head_text = fgl_winprompt(8,10, "Enter the new header name",
                                  " new name", 15, 0)
    CALL fgl_grid_header ("s_country_grid", "code_lit", head_text, "Left")

ON KEY (F12)
    LET head_text = fgl_winprompt(8,10, "Enter the new header name",
                                  " new name", 15, 0)
    CALL fgl_grid_header ("s_country_grid", "country_name",
                          head_text, "Left")

ON KEY (F1)

    LET arr_info =
"\n\n - The current row of the program array is: ||arr_curr(),
"\n\n - The size of the screen grid is: ||fgl_scr_size("s_country_grid"),
"\n\n - The size of the program array is: ||sizeof(a_country_list)

    CALL fgl_winmessage("Info", arr_info, "Info")

AFTER DISPLAY           -- opens a message box after the display is finished
    CALL fgl_winmessage("Congratulations!",
                        "You have seen the screen greed usage.",
                        "Exclamation")
END DISPLAY
CLOSE WINDOW w_countr_grid
CLEAR SCREEN
END FUNCTION

#####
# This function operates multidimensional arrays which are not
# arrays of records
#####

FUNCTION multi_dem_arr()
DEFINE

    a_one_dimensional_array  ARRAY[3] OF INTEGER,   -- one-dimensional array
    a_two_dimensional_array  ARRAY[3,3] OF CHAR(3), -- two-dimensional array
    a_three_dimensional_array ARRAY[2,2,2] OF VARCHAR(10)
                                         -- three-dimensional array

DISPLAY "Displaying elements of a single-dimensional array" AT 3,2
ATTRIBUTE(GREEN, BOLD)

# We initialize the elements of a_one_dimensional_array.
# As it has only one dimension and its size is 3, it can contain only
# 3 elements
    LET a_one_dimensional_array[1] = 1
    LET a_one_dimensional_array[2] = 2
    LET a_one_dimensional_array[3] = 3

```



```

DISPLAY "a_one_dimensional_array[1]"      = ", a_one_dimensional_array[1]
AT 4,2
DISPLAY "a_one_dimensional_array[2]"      = ", a_one_dimensional_array[2]
AT 5,2
DISPLAY "a_one_dimensional_array[3]"      = ", a_one_dimensional_array[3]
AT 6,2

# We initialize the elements of a_two_dimensional_array. As it has two
# dimensions each of which is of size 3, it can contain up to 3*3=9
# elements
LET a_two_dimensional_array[1,1] = "1,1"
LET a_two_dimensional_array[1,2] = "1,2"
LET a_two_dimensional_array[1,3] = "1,3"
LET a_two_dimensional_array[2,1] = "2,1"
LET a_two_dimensional_array[2,2] = "2,2"
LET a_two_dimensional_array[2,3] = "2,3"
LET a_two_dimensional_array[3,1] = "3,1"
LET a_two_dimensional_array[3,2] = "3,2"
LET a_two_dimensional_array[3,3] = "3,3"

DISPLAY "Displaying elements of a two-dimensional array" AT 8,2
ATTRIBUTE(GREEN, BOLD)

#We display only some of the elements
DISPLAY "a_two_dimensional_array[1,1]"      = ", a_two_dimensional_array[1,1]
AT 9,2
DISPLAY "a_two_dimensional_array[2,2]"      = ", a_two_dimensional_array[2,2]
AT 10,2
DISPLAY "a_two_dimensional_array[3,3]"      = ", a_two_dimensional_array[3,3]
AT 11,2

# We initialize the elements of a_three_dimensional_array
# This is a three-dimensional array with 2 as the size of each dimension.
# Thus the total number of elements in this array is 2*2*2=8
LET a_three_dimensional_array[1,1,1] = "1,1,1"
LET a_three_dimensional_array[1,1,2] = "1,1,2"
LET a_three_dimensional_array[1,2,1] = "1,2,1"
LET a_three_dimensional_array[1,2,2] = "1,2,2"
LET a_three_dimensional_array[2,1,1] = "2,1,1"
LET a_three_dimensional_array[2,1,2] = "2,1,2"
LET a_three_dimensional_array[2,2,1] = "2,2,1"
LET a_three_dimensional_array[2,2,2] = "2,2,2"

DISPLAY "Displaying elements of a three-dimensional array" AT 13,2
ATTRIBUTE(GREEN, BOLD)

# We display only some of the elements
DISPLAY "a_three_dimensional_array[1,1,1]"      = ,
a_three_dimensional_array[1,1,1] AT 14,2
DISPLAY "a_three_dimensional_array[2,1,1]"      = ,
a_three_dimensional_array[2,1,1] AT 15,2
DISPLAY "a_three_dimensional_array[2,2,2]"      = ,
a_three_dimensional_array[2,2,2] AT 16,2

DISPLAY "Displaying substrings of array elements" AT 18,2 ATTRIBUTE(GREEN, BOLD)

```



```
# You can use a substring operator([]) to display substrings of the array
# elements. Each element of array a_three_dimensional_array has
# 5 characters which are assigned above. The substring operator picks out
# only those which are specified.

LET v_char = a_three_dimensional_array[1,1,2][5]
DISPLAY "a_three_dimensional_array[1,1,2][5] = ",v_char AT 19,2
LET v_char = a_three_dimensional_array[2,1,1][2,4]
DISPLAY "a_three_dimensional_array[2,1,1][2,4] = ",v_char AT 10,2
LET v_char = a_three_dimensional_array[2,2,1][2,5]
DISPLAY "a_three_dimensional_array[2,2,1][2,5] = ",v_char AT 21,2

OPTIONS PROMPT LINE LAST - 1

LET answer = "" PROMPT "Press any key to see a screen array" FOR CHAR answer
ATTRIBUTE (YELLOW, BOLD)
CLEAR SCREEN

END FUNCTION

#####
# This function shows how dynamic arrays are declared and used. It also shows
# how the special methods are used for managing dynamic arrays
#####

FUNCTION dynamic_arr()
DEFINE dyn_arr_singl DYNAMIC ARRAY OF INT, -- a single-dimensional dynamic array
      dyn_arr_multi DYNAMIC ARRAY WITH 2 DIMENSIONS OF STRING,
                           -- a two-dimensional dynamic array
      i, ii INT

# Assigning values to a single dimensional array will resize it to the size
# of 10 elements
FOR i = 1 TO 10
    LET dyn_arr_singl[i] = i
END FOR

# Assigning values to a two-dimensional values will resize each of its
# dimensions to 20
FOR i = 1 TO 20

    FOR ii = 1 TO 20
        LET dyn_arr_multi[i,ii] = "dim 1 =",i," dim 2 =", ii
    END FOR
END FOR

DISPLAY "Some values of the two-dimensional array:" AT 3,2 ATTRIBUTE (GREEN,
BOLD)
DISPLAY "dyn_arr_multi[2,12] = ", dyn_arr_multi[2,12] AT 4,2
DISPLAY "dyn_arr_multi[11,7] = ", dyn_arr_multi[11,7] AT 5,2

DISPLAY "Some values of a single-dimensional array" AT 6,2 ATTRIBUTE (GREEN,
BOLD)
DISPLAY "dyn_arr_singl[2] = ",dyn_arr_singl[2] AT 7,2
DISPLAY "dyn_arr_singl[5] = ",dyn_arr_singl[5] AT 8,2
```



```
DISPLAY "Methods used with an array" AT 10,2 ATTRIBUTE (GREEN, BOLD)

# getsize() returns the current array size
DISPLAY "Array initial size (getsize()): ", dyn_arr_singl.getsize() AT 11,2

# Resizes the array to 30 elements, the last 20 elements will have 0 values
CALL dyn_arr_singl.resize(30)

DISPLAY "Array size after resize(30): ", dyn_arr_singl.getsize() AT 12,2

# Insert an element between element 1 and element 2 with value 200
CALL dyn_arr_singl.insert(2,200)

DISPLAY "An element was inserted - insert(2,200), now dyn_arr_singl[2] = ", dyn_arr_singl[2] AT 13,2

# Deletes all the elements from the array and sets its size to 0
CALL dyn_arr_singl.clear()

DISPLAY "The array size after clear() method: ", dyn_arr_singl.getsize() AT 14,2

OPTIONS PROMPT LINE LAST - 1

LET answer = "" PROMPT "Press any key to see a screen array" FOR CHAR answer
ATTRIBUTE (YELLOW, BOLD)
CLEAR SCREEN

END FUNCTION

#####
# This function assigns values to 50 elements of the array of records
# For each element (or a row) we need to assign three values, as the
# record has three members. Thus all in all we need to assign values to
# 50*3=150 elements. We transferred the initialization into a function,
# because it clutters up too much space.
#####

FUNCTION init_a_country_list()
    INITIALIZE a_country_list TO NULL

    LET a_country_list[1].country_id      = "004"
    LET a_country_list[1].code_lit       = "AFG"
    LET a_country_list[1].country_name   = "Afghanistan"
    LET a_country_list[2].country_id      = "008"
    LET a_country_list[2].code_lit       = "ALB"
    LET a_country_list[2].country_name   = "Albania"
    LET a_country_list[3].country_id      = "010"
    LET a_country_list[3].code_lit       = "ATA"
    LET a_country_list[3].country_name   = "Antarctica"
    LET a_country_list[4].country_id      = "012"
    LET a_country_list[4].code_lit       = "DZA"
    LET a_country_list[4].country_name   = "Algeria"
    LET a_country_list[5].country_id      = "016"
```



```
LET a_country_list[5].code_lit = "ASM"
LET a_country_list[5].country_name = "American Samoa"
LET a_country_list[6].country_id = "020"
LET a_country_list[6].code_lit = "AND"
LET a_country_list[6].country_name = "Andorra"
LET a_country_list[7].country_id = "024"
LET a_country_list[7].code_lit = "AGO"
LET a_country_list[7].country_name = "Angola"
LET a_country_list[8].country_id = "028"
LET a_country_list[8].code_lit = "ATG"
LET a_country_list[8].country_name = "Antigua And Barbuda"
LET a_country_list[9].country_id = "031"
LET a_country_list[9].code_lit = "AZE"
LET a_country_list[9].country_name = "Azerbaijan"
LET a_country_list[10].country_id = "032"
LET a_country_list[10].code_lit = "ARG"
LET a_country_list[10].country_name = "Argentina"
LET a_country_list[11].country_id = "036"
LET a_country_list[11].code_lit = "AUS"
LET a_country_list[11].country_name = "Australia"
LET a_country_list[12].country_id = "040"
LET a_country_list[12].code_lit = "AUT"
LET a_country_list[12].country_name = "Austria"
LET a_country_list[13].country_id = "044"
LET a_country_list[13].code_lit = "BHS"
LET a_country_list[13].country_name = "Bahamas"
LET a_country_list[14].country_id = "048"
LET a_country_list[14].code_lit = "BHR"
LET a_country_list[14].country_name = "Bahrain"
LET a_country_list[15].country_id = "050"
LET a_country_list[15].code_lit = "BGD"
LET a_country_list[15].country_name = "Bangladesh"
LET a_country_list[16].country_id = "051"
LET a_country_list[16].code_lit = "ARM"
LET a_country_list[16].country_name = "Armenia"
LET a_country_list[17].country_id = "052"
LET a_country_list[17].code_lit = "BRB"
LET a_country_list[17].country_name = "Barbados"
LET a_country_list[18].country_id = "056"
LET a_country_list[18].code_lit = "BEL"
LET a_country_list[18].country_name = "Belgium"
LET a_country_list[19].country_id = "060"
LET a_country_list[19].code_lit = "BMU"
LET a_country_list[19].country_name = "Bermuda"
LET a_country_list[20].country_id = "064"
LET a_country_list[20].code_lit = "BTN"
LET a_country_list[20].country_name = "Bhutan"
LET a_country_list[21].country_id = "068"
LET a_country_list[21].code_lit = "BOL"
LET a_country_list[21].country_name = "Bolivia"
LET a_country_list[22].country_id = "070"
LET a_country_list[22].code_lit = "BIH"
LET a_country_list[22].country_name = "Bosnia And Herzegovina"
LET a_country_list[23].country_id = "072"
LET a_country_list[23].code_lit = "BWA"
```



```
LET a_country_list[23].country_name = "Botswana"
LET a_country_list[24].country_id = "074"
LET a_country_list[24].code_lit = "BVT"
LET a_country_list[24].country_name = "Bouvet Island"
LET a_country_list[25].country_id = "076"
LET a_country_list[25].code_lit = "BRA"
LET a_country_list[25].country_name = "Brazil"
LET a_country_list[26].country_id = "084"
LET a_country_list[26].code_lit = "BLZ"
LET a_country_list[26].country_name = "Belize"
LET a_country_list[27].country_id = "086"
LET a_country_list[27].code_lit = "IOT"
LET a_country_list[27].country_name = "British Indian Ocean Territory"
LET a_country_list[28].country_id = "090"
LET a_country_list[28].code_lit = "SLB"
LET a_country_list[28].country_name = "Solomon Islands"
LET a_country_list[29].country_id = "092"
LET a_country_list[29].code_lit = "VGB"
LET a_country_list[29].country_name = "Virgin Islands (British)"
LET a_country_list[30].country_id = "096"
LET a_country_list[30].code_lit = "BRN"
LET a_country_list[30].country_name = "Brunei Darussalam"
LET a_country_list[31].country_id = "100"
LET a_country_list[31].code_lit = "BGR"
LET a_country_list[31].country_name = "Bulgaria"
LET a_country_list[32].country_id = "104"
LET a_country_list[32].code_lit = "MMR"
LET a_country_list[32].country_name = "Myanmar"
LET a_country_list[33].country_id = "108"
LET a_country_list[33].code_lit = "BDI"
LET a_country_list[33].country_name = "Burundi"
LET a_country_list[34].country_id = "112"
LET a_country_list[34].code_lit = "BLR"
LET a_country_list[34].country_name = "Belarus"
LET a_country_list[35].country_id = "116"
LET a_country_list[35].code_lit = "KHM"
LET a_country_list[35].country_name = "Cambodia"
LET a_country_list[36].country_id = "120"
LET a_country_list[36].code_lit = "CMR"
LET a_country_list[36].country_name = "Cameroon"
LET a_country_list[37].country_id = "124"
LET a_country_list[37].code_lit = "CAN"
LET a_country_list[37].country_name = "Canada"
LET a_country_list[38].country_id = "132"
LET a_country_list[38].code_lit = "CPV"
LET a_country_list[38].country_name = "Cape Verde"
LET a_country_list[39].country_id = "136"
LET a_country_list[39].code_lit = "CYM"
LET a_country_list[39].country_name = "Cayman Islands"
LET a_country_list[40].country_id = "140"
LET a_country_list[40].code_lit = "CAF"
LET a_country_list[40].country_name = "Central African Republic"
LET a_country_list[41].country_id = "144"
LET a_country_list[41].code_lit = "LKA"
LET a_country_list[41].country_name = "Sri Lanka"
```



```
LET a_country_list[42].country_id = "148"
LET a_country_list[42].code_lit = "TCD"
LET a_country_list[42].country_name = "Chad"
LET a_country_list[43].country_id = "152"
LET a_country_list[43].code_lit = "CHL"
LET a_country_list[43].country_name = "Chile"
LET a_country_list[44].country_id = "156"
LET a_country_list[44].code_lit = "CHN"
LET a_country_list[44].country_name = "China"
LET a_country_list[45].country_id = "158"
LET a_country_list[45].code_lit = "TWN"
LET a_country_list[45].country_name = "Taiwan, Province Of China"
LET a_country_list[46].country_id = "162"
LET a_country_list[46].code_lit = "CXR"
LET a_country_list[46].country_name = "Christmas Island"
LET a_country_list[47].country_id = "166"
LET a_country_list[47].code_lit = "CCK"
LET a_country_list[47].country_name = "Cocos (Keeling) Islands"
LET a_country_list[48].country_id = "170"
LET a_country_list[48].code_lit = "COL"
LET a_country_list[48].country_name = "Colombia"
LET a_country_list[49].country_id = "174"
LET a_country_list[49].code_lit = "COM"
LET a_country_list[49].country_name = "Comoros"
LET a_country_list[50].country_id = "175"
LET a_country_list[50].code_lit = "MYT"
LET a_country_list[50].country_name = "Mayotte"

END FUNCTION
```

Form Files

The following form file called "countr_lst.per" is used in this application:



```
\g|          [f5      ]      [f6      ]      [f7      ]      [\g
\g|\g      ]      [f6      ]      [f7      ]      [\g
\g|          [f5      ]      [f6      ]      [f7      ]      [\g
\gb-----d\g

}

ATTRIBUTES
f0 = formonly.headr, config = "    ", widget = "label", center;
f1 = formonly.countrylist, config = "Country List", widget = "label";
f00 = formonly.country_id TYPE CHAR;
f01 = formonly.code_lit TYPE CHAR;
f02 = formonly.country_name TYPE CHAR;
f2 = formonly.num, config = "Num", widget = "label";
f3 = formonly.lit, config = "Lit", widget = "label";
f4 = formonly.label, config = "Country", widget = "label";
f5 = formonly.accpt, config = "OK {OK}", widget = "button", ;
f6 = formonly.info, config = "F1 {Info}", widget = "button", ;
f7 = formonly.cancl, config = "F10 {Cancel}", widget = "button", ;

INSTRUCTIONS
    DELIMITERS "  "
    SCREEN RECORD s_country_list[10] (country_id THRU country_name)
```

The second form file used in this application is called scr_grid.per

```
DATABASE formonly
SCREEN SIZE 26 BY 80
{
[f0
\gp-----[f1 ]-----q\g
\g|
\g|\f00|f01|f02
\g| [f03 ] [f04 ] [f05 ] | \g
\g|
\gb-----d\g
}

ATTRIBUTES
f0 = formonly.headr, config = "    ", widget = "label", center;
f1 = formonly.countrylist, config = "Country List", widget = "label";
```



```
f00 = formonly.country_id TYPE CHAR, LEFT,
      OPTIONS = "HEADER = 'Num', KEY F10"; -- specify the column header
                           -- as 'Num' and bind the column
                           -- with the F9 key
f01 = formonly.code_lit TYPE CHAR, LEFT,
      OPTIONS = "HEADER = 'Lit', KEY F11";-- specify the column header
                           -- as 'Lit' and bind the column
                           -- with the F10 key
f02 = formonly.country_name TYPE CHAR, LEFT,
      OPTIONS = "HEADER = 'Country', KEY F12";-- specify the column header
                           -- as 'Country' and bind the
                           -- column with the F11 key
f03 = formonly.head, config ="F9 {Change Headers}", widget ="button";
f04 = formonly.info, config ="F1 {Info}", widget ="button";
f05 = formonly.acc, config ="ACCEPT {Accept}", widget ="button";
```

INSTRUCTIONS

```
DELIMITERS " "
SCREEN GRID s_country_grid[10] (country_id THRU country_name)
```



Performing Input

The program ability to operate the data entered by the user is very important. It makes program operation more accurate and makes the program itself more multipurpose. We have already discussed the simplest means of user input - the [PROMPT](#) statement. Unfortunately, this statement doesn't always meet all the demands concerning the format and the size of the input value. It cannot also be used for passing values from screen forms.

In this chapter we will introduce the basic idea of the INPUT statement, which is a powerful and convenient means of data input performed into form fields.

The INPUT Statement

The INPUT statement is used to let the user enter one or more values into fields of a screen form and to pass the entered values to specified program variables. The syntax of the INPUT statement is as follows:

```
INPUT Binding_clause [ATTRIBUTE clause] [HELP number] [INPUT CONTROL BLOCK]
```

The INPUT statement links fields of a form file with variables used to store the values entered in these fields. During the INPUT the fields which it references are activated. When the program executes an INPUT statement, it displays default values, if any, to the fields of the screen form, positions the cursor to the first field and waits for the user to enter a value to this field. When the user presses the Accept key or moves the cursor to the next field, the program assigns the entered value to the corresponding program variable.

The Binding Clause

The Binding clause creates temporary bounds between form fields and 4GL variables. This manipulation is necessary in order to let the program pass the user-entered values to program variables and to process them. The syntax of the binding clause is as follows:

```
Variable_list FROM Field_list
```

The variable list must include at least one variable of any data type. If you add several variables to the variable list, you have to separate them with commas. The field list consists of the field names. They can be preceded with the names of screen records, which can be the default *formonly* record:

```
INPUT char1, integ1 FROM formonly.char_val1, formonly.integ_val2
```

```
INPUT char2 FROM f001
```

The form fields specified in the INPUT statement become active and available for input, when 4GL executes the corresponding INPUT statement. While each INPUT statement is processed separately, even though the three form fields in the example above are located within the same form, only two become active at first. The last field becomes active when the input for the first two is finished and they are no more available for input.



The number of the items that comprise the variable list must match the number of the fields listed in the field clause. The field and the corresponding variable must match in data type or be of compatible data types. If the value entered to a field and the field are of an incompatible data types, 4GL displays a warning message and waits for the user to enter a valid value. If the user enters no value to a field and presses the Accept key, the variable connected to the field retains its initial value.

You can input the values to a program record from a screen record if the number of program and screen record members and their data types match. In this case, the following structure can be applied:

```
INPUT program_record.* FROM screen_record.*
```

If you want the user to input values for only a part of program record members, you can use the *first* THRU (or THROUGH) *last* structure, which will implicitly specify record members that are listed between the *first* and the *last* variables inclusive:

```
DEFINE
  rec RECORD
    mem1, mem2, mem3 INTEGER,
    mem4, mem5, mem6 CHAR (5)
  END RECORD

  ...
  ...
INPUT rec.mem2 THROUGH rec.mem4 FROM formonly.f001, formonly.f002, formoly.f003
```

In this example, the user will have to enter values for variables *mem2*, *mem3*, and *mem4* which are the members of a program record *rec*.

However, the THRU keyword cannot appear in the FROM clause to specify a set of contiguous form fields.

When the program executes the INPUT statement, the cursor moves from field to field in the order, which corresponds to the order of field names listed in the field clause, unless other order is specified.

The ATTRIBUTE Clause

The ATTRIBUTE clause is used to apply a set of attributes to the fields from which the input is performed. The attributes which can be specified here are the same as the other text attributes, described for the [DISPLAY statement](#).

The attributes specified in the INPUT statement temporary override those specified previously in OPTIONS, OPEN WINDOW, or DISPLAY FORM statements. They come in effect when the form is activated and the user works with it. When the user stops working with the form, the INPUT attributes become deactivated, and the previous attributes are applied.

This is an example of the ATTRIBUTE clause application:

```
INPUT m_name FROM formonly.f_name ATTRIBUTE (GREEN)
```

The value entered by the user to the field *f_name* will be displayed in green during the input.



The HELP Clause

The HELP clause is used to specify the number of a help message that is to be displayed to the help window. The help window containing a message appears when the user presses the Help key (CONTROL-w by default) with the cursor being positioned in one of the form fields, listed in the INPUT statement.

The HELP clause follows the same rules which have already been described for [other](#) statements supporting this clause.

The INPUT Control Block

The INPUT control block is optional and is used to control the INPUT statement execution. The block must consist of one or more statements and an activation clause which specifies the conditions of the statements execution. You can include the following items to the INPUT control block:

- The BEFORE INPUT and AFTER INPUT clauses - contain the statements that are to be executed before or after the INPUT statement execution;
- The BEFORE FIELD and AFTER FIELD clauses - contain the statements that are to be executed before or after the user enters values to specified screen fields;
- The ON KEY clause - contains statements that are to be executed when the user presses a combination of keys;

The above listed clauses can contain some special keywords apart from the normal 4GL statements, they are:

- NEXT FIELD - specifies the field to which the cursor is to be moved after the current INPUT control clause finishes executing.
- EXIT INPUT - terminates the input
- CONTINUE INPUT - resumes input

If you add any clauses to the control block, you have to put END INPUT keywords after the last one of them in order to separate the statements in control block clauses from the statements that are to be executed after the INPUT block:

```
INPUT a, b FROM formonly.f_a, formonly.f_b
  BEFORE INPUT...
  ON KEY ...
  AFTER FIELD...
END INPUT
```

The clauses of the INPUT statement are executed in the following order regardless of the order they are listed in:

- BEFORE INPUT
- BEFORE FIELD - for the current field, it can be executed a number of times, before the user is allowed to enter information in any field with the BEFORE FIELD clause
- ON KEY - if the activation key is pressed



- AFTER FIELD - executed as many times as the cursor leaves a fields to which this clause is assigned
- AFTER INPUT

If the user interrupts the input by pressing the Accept key 4GL still executes first the AFTER FIELD clause for the current field, if there is one, and then the AFTER INPUT clause, if it is present. Thus if any of these clauses contains statements that make 4GL continue the input, the input will not be terminated.

The BEFORE INPUT Block

The BEFORE INPUT block is used to specify the actions that will be performed before the user is allowed to enter values to form fields:

```
INPUT p_auth.* FROM s_auth.*  
    BEFORE INPUT  
        PROMPT "Enter your name" FOR u_name  
    ...  
END INPUT
```

This snippet of a source code will make the program display a prompt line and wait for the user to enter his name before the INPUT statement is executed and the form fields are activated.

The BEFORE FIELD Block

The BEFORE FIELD block is used to specify actions that are to be performed before the user is allowed to enter anything to the specified field. It is executed when the cursor is moved to the field specified in the BEFORE FIELD clause, but before the field is activated for the input.

The BEFORE FIELD keywords must be immediately followed by one or more field names. If there are more than one field name, they should be separated by commas:

```
BEFORE FIELD field_name [,field_name...]  
    statement_list
```

The INPUT statement may include one or several BEFORE FIELD blocks, but only one block can be specified for each field. This part of the source code makes the program prompt for some values before the user can enter data to form fields:

```
INPUT s_name, s_age, s_child FROM formonly.name, formonly.age,formonly.child  
    BEFORE FIELD s_name  
        PROMPT "Are you Sir or Madam?" FOR s_m  
    BEFORE FIELD s_child  
        PROMPT "Are you married?(Y/N)" FOR wedlock  
    ...  
END INPUT
```

The ON KEY Block

The ON KEY block is used to specify some actions that are to be performed when the user presses the keys listed in this block. The keywords that can be specified here and the rules of key specification are described for the [PROMPT](#) statement.



The following example specifies actions for the CONTROL-F key:

```
OPEN FORM my_f FROM "personal"  
  
DISPLAY FORM my_f  
  
INPUT pers.* FROM person.*  
  
ON KEY (CONTROL-f)  
  
OPEN WINDOW educ WITH FORM "education"  
  
INPUT educ.* FROM educat.*  
  
CLOSE WINDOW educ  
END INPUT
```

These lines open a form and activate the INPUT statement for some of the form fields. When the user presses CONTROL-f, the program opens a new window and activates another INPUT statement for another form file. When this statement is executed, the program closes the new window and the user can continue input to the primary form.

When the user presses one of the *keys*, specified in the ON KEY block, the program performs the following actions:

- Suspends the current input;
- Saves the values that have already been entered by the user;
- Executes the statements, specified in the activated ON KEY block;
- Re-displays the values that have been entered by the user to the form fields;
- Positions the cursor to the field where it had been before the *key* were pressed.

Unlike the PROMPT statement, the ON KEY clause of the INPUT statement has an optional *infield()* operator, which allows the programmer to trigger field-specific actions. This extension is applicable only for Lycia II and above. Its syntax is as follows:

```
ON KEY(key1 [,key2...]) INFIELD(field1 [,field2...])
```

The ON KEY clause with the *infield()* operator will be executed, only if the cursor is located in one of the fields specified in the *infield()* operator, when the referenced key is pressed. In other cases such ON KEY clause will be ignored. If there is another ON KEY clause referencing the same key which has no *infield()* operator, it will be executed then the cursor is outside of the fields specified in the *infield()* operator, like in the example below:

```
INPUT a,b,c,d FROM f1,f2,f3,f4  
ON KEY(F5)  
CALL fgl_message_box( "Global ON KEY event" )  
ON KEY(F5) INFIELD(F2,F3)
```



```
CALL fgl_message_box( "ON KEY event for fields f2 and f3" )
END INPUT
```

The AFTER FIELD Block

The AFTER FIELD block is used to specify the actions that will be performed when the cursor leaves the specified field. The syntax is similar to that of the BEFORE FIELD clause:

```
AFTER FIELD field_name [,field_name...]
statement_list
```

The cursor leaves the field when the user presses the HOME or END keys, any arrow key (RIGHT arrow key when the cursor is positioned after the last entered character, LEFT arrow key when the cursor is positioned before the first entered character), the RETURN, TAB, or Accept key, the Interrupt or Quit key (if the DEFER statement prevents the program from being terminated by these keys).

Only one AFTER FIELD block can be specified for each field. The following AFTER FIELD block checks the user access to the program menu:

```
MENU "MAIN"
COMMAND "Go" "Press to Start"

OPEN FORM inp FROM "Input_form"

DISPLAY FORM inp

INPUT dat.* FROM purp./*

# the following clause specifies the actions to be
# performed after the login field is filled

AFTER FIELD purp.login

IF log NOT MATCHES pass -- checks the entered value
THEN HIDE OPTION "Settings"

END IF

END INPUT
COMMAND "Settings"
...
END MENU
```

The AFTER INPUT Block

The AFTER INPUT block is used to assign the actions that will be performed by the program when all the form fields specified in the INPUT statement are filled and the user presses the Accept key. The AFTER INPUT block can be used to evaluate, check, change, or process the values entered by the user.



The AFTER INPUT block is executed only if the user presses Accept key and terminates the INPUT statement. The AFTER INPUT statement does not come into effect if the EXIT INPUT keywords are executed.

The Field Order

By default, the cursor movement across the screen is determined by the order in which the field names are listed in the INPUT statement. If no other options are specified, the cursor moves to the next field, when the user presses the DOWN and RIGHT arrow keys, the TAB, and the RETURN key. The cursor moves to the previous field when the user presses the UP and LEFT arrow keys.

You can explicitly assign the field to which the cursor will be moved by adding the NEXT FIELD clause to one of the control clauses of an INPUT statement. 4GL supports three options that can follow the NEXT FIELD keywords:

- NEXT FIELD NEXT means that the cursor will move to the next field (just like the default option);
- NEXT FIELD PREVIOUS means that the cursor is to be positioned in the previous field;
- NEXT FIELD *field-name* specifies the name of the field to where the cursor is to be moved.

Here is an example of the NEXT FIELD keywords application in the AFTER FIELD block of the INPUT statement:

```
INPUT mem.* FROM staff.*  
  
        AFTER FIELD us_name  
  
                IF us_name matches " " -- checks if any value was entered  
  
                        THEN NEXT FIELD us_name --if no value was entered,  
                                         --re-activates the field  
  
                END IF  
  
                ...  
        END INPUT
```

The NEXT FIELD clause may appear in any INPUT control block. It is typically used in conditional statements, and it *must* be used in a conditional statement, if it appears within the AFTER INPUT block. Otherwise, it deprives the user of the capability to exit the form. If you want to restrict the access to the field, you can use the NEXT FIELD keywords in the BEFORE FIELD block.

The CONTINUE INPUT Keywords

The CONTINUE INPUT keywords can appear in any of the INPUT control clauses. When 4GL encounters these keywords, it skips all the subsequent statements of this clause and returns program control to the most recently occupied field of the current form.

These keywords may prove useful, if you need to return program control to a user from a deeply nested control clause. It can also be used in the AFTER INPUT clause in order to return the cursor to the form after examining the entered values. You should not use it in an AFTER INPUT clause outside a conditional statement otherwise the user will not be able to complete the input.



The EXIT INPUT Keywords

The EXIT INPUT keywords terminate the INPUT statement. It can be located in any control clause. When 4GL encounters these keywords, it skips all the statements between them and the END INPUT keywords, deactivates the form and continues execution from the first statement following the END INPUT keywords.

Facilitating Input

There are several built-in functions and statements which can facilitate the processing of the input. They are discussed in full detail in the following chapters, whereas this section only introduces some of them.

Fgl_lastkey()

The *fgl_lastkey()* function is used to return an integer code of the last key pressed by the user during the value input to the form field. The *last* key means the key that the user pressed the last before the *fgl_lastkey()* function was invoked. This function does not require arguments as well. The syntax of the function invocation is:

```
fgl_lastkey()
```

Fgl_keyval()

Function *fgl_keyval()* accepts the name of a key as an argument and returns the code which is equal to the code returned by the *fgl_keyval()* function. These functions can be used in a Boolean expression to confirm whether the key pressed by the user corresponds to the required key, as illustrated in the demo example.

```
fgl_keyval("down")
```

The ERROR Statement

The ERROR statement displays its message at the error line which is typically the last line of the screen. Thus it is more convenient than the DISPLAY statement in those cases when you are not sure where to display a message, as it may overlap something. The ERROR keyword must be followed by the quoted message:

```
ERROR "This is an error"
```

Input Field Attributes

In the previous chapters we have discussed some of the field attributes and general attribute syntax. For now we have only used the attributes which take effect regardless of whether a field is used for display or for input. This chapter will dwell upon the attributes which have the visible effect only when a field is used for input. They will not cause any errors, if a field is not used for input, but they will also have no effect.



AUTONEXT

The AUTONEXT attribute moves the cursor to the next field when the value of sufficient size is entered to the current field without the RETURN key pressed. The sufficient size is the size determined by the form field delimiters.

The attribute doesn't need any additional identifiers or specifications. Here is an example of the AUTONEXT attribute specification:

```
pos_cod = formonly.ind, AUTONEXT;
```

The cursor will move to the next field as soon as the field tagged as *pos_cod* becomes full.

The AUTONEXT attribute is useful when you input values to character fields which require data of a standard length, such as numeric telephone numbers, postal codes, etc. It can also be useful if the field supports only one-character values, so the cursor will move to the next field as soon as the character is entered.

If the data entered to the field that has the AUTONEXT attribute doesn't correspond to the required format, the cursor won't move to the next field until the mistake is corrected.

CENTURY

The CENTURY attribute is used to specify how a program should expand one- and two-digit year parts of DATE and DATETIME fields.

The attribute is similar to the CENTURY attribute of the PROMPT statement. The syntax of the attribute is:

```
CENTURY = "Symbol"
```

Where *symbol* stands for one of the symbols that specify the algorithm of expansion. The symbols that can be specified for the CENTURY attribute are:

Symbol	Meaning	Algorithm of Abbreviated Years Expansion
C or c	Closest	Use the year that is closest to the current date, regardless whether it will be past, current, or future year
F or f	Future	The nearest year is the future is used
P or p	Past	The nearest year in past is used
R or r	Current	The current year is used

The following form field attributes will make the program add the current century to a DATE value if it is represented by one or two symbols:

```
curr_date = formonly.curr_date, TYPE DATE, CENTURY = "R";
```

COMMENTS

You can specify comment messages for form fields which can contain information about the purpose of the field, restrictions, etc. The COMMENTS attribute is used to display a message to the COMMENT line of the screen when the cursor is positioned to the specified form field. When the cursor moves to another field, the message disappears or is replaced by another COMMENT, specified for the next field.



The syntax of the COMMENTS attribute is:

```
COMMENTS = "message"
```

The COMMENTS attribute is usually applied in order to give the user a piece of information or some instructions, for example, to explain what kind of values can be input into the field. The comments are displayed automatically when the cursor is in the corresponding field, the user doesn't need to press any keys to see them.

The example below specifies a message describing what values can be entered to the field:

```
f001=formonly.d_date, TYPE DATE, COMMENTS = "Enter the date";
```



Note: A 4GL program may use the same form file for several tasks. Use the COMMENTS attribute only if the field has the same purpose in all the tasks, because a field comment cannot be changed at runtime.

Comment Line Position on the Screen

The default position of the Comments line is LAST for the 4GL window, and line 23 for the 4GL screen (which is the last but one).

Its position can be changed by means of the OPTIONS statement (which will change the comments line position for all the 4GL windows and the 4GL screen), or within the ATTRIBUTE clause of the OPEN WINDOW statement (if you want to specify a new place for the Comment line in the new window only):

```
MAIN

DEFINE ...

OPTIONS COMMENT LINE LAST-1 -- the comment line for all the 4GL
-- Windows that will be opened by
-- this program is specified here

...

OPEN WINDOW my_win AT 3,6 WITH 10 ROWS, 60 COLUMNS

ATTRIBUTE (BORDER, COMMENT LINE 2) -- the comment line of this
-- window is specified here
```

You can use COMMENT LINE OFF keywords to hide the comment line even if there are comments specified for some form fields. If you use the COMMENT LINE OFF keywords as a part of the OPTIONS statement in the MAIN program block, the comments will be hidden for all the forms opened by the program. If you use these keywords as an attribute of the OPEN WINDOW statement, the program will hide only the comment line of the form opened in this window.



To show the hidden comment line, use the COMMENT LINE ON keywords. It can also be used as an OPTION or as an OPEN WINDOW statement attribute.

In the following example, all the comments are hidden, but they are allowed for the form file opened in window *win_1*:

```
MAIN

DEFINE ...

OPTIONS COMMENT LINE OFF -- hides all the comment lines

...

OPEN WINDOW win_1 AT 3,6 WITH FORM "formfile_1"

ATTRIBUTE (BORDER, COMMENT LINE ON) -- the comment line will be
-- visible in this window

...
```

DEFAULT

The DEFAULT attribute is used to assign a default value to a field. This value will be displayed in the field and can be replaced by another value during the input. This value does not influence the value of the variable used for input directly. However, if the user does not enter another value and presses the Accept key, it will be assigned to the variable as any other input value.

The syntax of the DEFAULT attribute is as follows:

```
DEFAULT=value
```

The *value* format depends on the TYPE of the form field. It can be represented by a quoted string, a literal number, INTERVAL, DATE, or DATETIME value.

When you assign a value for the DEFAULT attribute, you should remember the following rules:

- You don't have to quote a character string unless it contains blank spaces, commas, any special characters or does not begin with a letter. If the character string is not enclosed in quotes, the program downshifts all the upper-case letters.

```
f01=formonly.country TYPE CHAR, DEFAULT = Australia;
```

- The DEFAULT value of a DATE field should be enclosed in quotation marks.

```
f02=formonly.b_date TYPE DATE, DEFAULT = "08/25/1979";
```

- The DEFAULT value of a DATETIME or INTERVAL field can be quoted or entered as a literal without quotes.



```
f03=formonly.time_f TYPE, DATETIME = DATETIME (2010-06-14 12:58:47.56)
YEAR TO FRACTION(2)
```

The default value can also be returned by an operator or a built-in function, if the data type of the returned value is compatible with the data type of the field. In this case, the arguments and operands must be represented by literal values or by the constants TRUE or FALSE. For example, a default value of DATE data type can be assigned as:

```
dt = formonly.dat TYPE DATE, DEFAULT = CURRENT;
```

The CURRENT operator will return the current date and time as a DEFAULT value of a DATETIME field.

Here is an example of the DEFAULT attribute applied to a form field:

```
country=formonly.count, DEFAULT = "DE", AUTONEXT;
```

You can disable default values by using the WITHOUT DEFAULTS option of the INPUT statement. The WITHOUT DEFAULTS keywords must be placed right after the variable list of the INPUT statement:

```
INPUT char1, char2 WITHOUT DEFAULTS FROM formonly.char1, formonly.char2
```

This line will prevent the program from displaying the default values to the form fields *char1* and *char2*.

FORMAT

The FORMAT attribute can be applied to DECIMAL, FLOAT, SMALLFLOAT, or DATE fields in order to influence the format of the value in the field. The syntax of the FORMAT attribute is:

```
FORMAT = "format string"
```

The Format String for Numeric Values

When the FORMAT attribute is applied to a field which supports numeric values, the digits in the "format string" are represented by sharp symbols (#) and are separated with a decimal point:

```
FORMAT = "####.###"
```

This attribute allows the field to include at least four digits to the left and three symbols to the right from the decimal point.

It is important, that the number of the symbols to the left of the point indicates the *minimum* number of digits that comprise the whole part of the number; the number of symbols to the right form the point indicates the precise number of decimal places.

If the whole part of the value is shorter than it is required by the FORMAT attribute, the value is prefixed by blank spaces and right-aligned. If the decimal value entered by the user is larger than it is specified in the FORMAT attribute, 4GL rounds the value before it is displayed to the field.

The locale settings can affect how the values entered to the field are evaluated. In English locale, the period symbol (.) indicates the decimal separator, and the comma symbol (,) stands for the thousands separator.



These meanings may be different in other locales. For example, if the field format is set as `##,###.##`, the US English locale will recognize the value `12345.67` as `12,345.67`, whereas the German locale will treat it as `12.345,67`.

The FORMAT for DATE Values

When the FORMAT attribute is applied to DATE values, the *format-string* can contain the following symbols:

Symbol	Result
<code>yyyy</code>	A four-digit representation of the year
<code>yy</code>	A two-digit representation of the year, which includes the decade and the year of the decade
<code>mmm</code>	A month is represented by a three-letter abbreviation, e.g., Mar for March, Apr for April
<code>mm</code>	A month is represented by two digits, for example, 03 for March, 08 for August.
<code>ddd</code>	A day of the week is represented by a three-letter abbreviation, e.g., Sun for Sunday, Mon for Monday
<code>dd</code>	A day of the month is represented by a two-digit integer ranging from 01 to 31

You can use the FORMAT attribute to specify any display format you want. If you use characters and symbols that are not listed in the table above, 4GL will treat them as literals and display them within the value.

When you use the FORMAT attribute, you have to enter a literal value to the field, and the program will convert it automatically to the specified format.

Here are examples of different *format-strings* used within the FORMAT attribute. The value displayed corresponds to the 16th of July, 2010:

FORMAT	Result
No FORMAT	07/16/2010
FORMAT = "yyyy-mm-dd"	2010-07-16
FORMAT = "mm-dd-yy, ddd"	07-16-2010, Fri
FORMAT = "ddd, dd mmm, yyyy"	Fri, 23 Jul, 2010

INVISIBLE

The INVISIBLE attribute is used to make the data entered by the user invisible. This keyword cannot appear in the COLOR field attribute, it is used as an independent attribute instead. It does not acquire any additional keywords.

When the INVISIBLE attribute is applied, the value entered into the field by the user can't be seen, but the value is nevertheless received by the 4GL:

```
pass = formonly.passw TYPE CHAR, INVISIBLE,
COMMENTS = "Enter your password";
```

The INVISIBLE attribute doesn't affect other aspects of value entry and processing. It doesn't make the entered value inaccessible for the program and doesn't prevent it from being displayed by means of the



DISPLAY statement. It doesn't hide the value which was displayed to the field by means of the DISPLAY, DISPLAY FORM, and other displaying statements.

If a field has some display attributes (i.e. COLOR attribute) besides the INVISIBLE attribute, the latter is ignored.

NOENTRY

The NOENTRY attribute is used to prevent data entry when the INPUT statement is executed. The field that has the NOENTRY attribute can be included to the INPUT statement, but the program won't position the cursor to this field.

You can use the attribute when the field gets its value not from the user, but from the program. It is convenient to apply this attribute to a member field of a screen record or array when you are going to input values by means of the structures like:

```
INPUT program_record.* FROM screen_record.*
```

This will prevent you from the necessity to list all the fields except the one you do not need to receive the input.

PICTURE

The PICTURE attribute is used to specify a character pattern applied to the data entered to a character field.

The syntax of the PICTURE attribute is as follows:

```
PICTURE = "format-string"
```

A *format-string* can include three special symbols: **A** stands for any letter, **#** stands for any digit, and **X** stands for any character. All the other characters used within the format-string, will be treated as literals. When the field with the PICTURE attribute is displayed, 4GL displays literal characters to it and leaves blank spaces instead of the special symbols.

For example, such field specification:

```
char1 = formonly.char1, PICTURE = "Passport: AA-XXX-XXX";
```

Will produce the following field display:

```
[Passport: - - ]
```

The user will have to enter two character symbols to the first gap and three-digit strings into each of the following gaps. The PICTURE attribute automatically prevents the user from entering the values which do not correspond to the specified pattern. If the user enters a character that does not correspond to the format-string, the character is not displayed, and the cursor doesn't move forward until a valid character is entered.



	<p>Note. The number of characters in the <i>format-string</i> specification must match the length of the field. Otherwise, a compile-time error will occur.</p> <p>However, the user doesn't have to enter the entire string when the INPUT statement is executed.</p>
---	---

If some symbols (hyphens, parentheses, points, commas, etc) are the part of the format string, the user doesn't have to enter them. They are displayed automatically.

REQUIRED

The REQUIRED attribute is used to force the user to enter data when the INPUT statement is in effect. The user will not be allowed to leave the field until they enter some value.

The REQUIRED keyword may be useful, when the field name is listed in the INPUT statement and the data entry to the field is obligatory.

```
char1 = formonly.char1, REQUIRED;
```

You shouldn't specify a default value for the required field. If you do, the program will treat the default value as the entered one and satisfy the REQUIRED attribute automatically, so that it will be of no use, as the result.

The attribute requires the user to enter printable characters to the field. If the user enters the value and then erases it, 4GL treats the REQUIRED attribute as a satisfied one.

Form Tabs

Lycia supports forms that have several pages with different widgets on them. A page of the form is named a *form tab*.

When you create a form with several tabs and display it to the screen, it looks as follows:



This screenshot shows a Windows application window titled "w1". At the top, there are three buttons: a green checkmark labeled "Accept", a red X labeled "Cancel", and a yellow question mark labeled "Help". Below these are two tabs: "Tab_1" and "Tab_2", with "Tab_1" being the active tab. The main area contains four empty rectangular input fields arranged vertically. A horizontal line separates this from a large empty space. At the bottom right of the window is a toolbar with several icons: a square, a circle, a triangle, a double arrow, and a small icon with a diagonal line. The window has standard Windows-style borders and control buttons.

In this form, all the fields are empty. You can see two tabs at the top of the form. These tabs indicate what page of the form is active now. In the picture above, the tab *Tab_1* is active. If *Tab_2* becomes active, you will see the following picture:

This screenshot shows the same Windows application window "w1" as the previous one, but with "Tab_2" active instead of "Tab_1". The main area now displays the Querix logo, which consists of a stylized molecular structure made of spheres in grey, blue, purple, and red, positioned above the word "Querix" in a large, bold, blue font. The four empty rectangular input fields from the previous screen are still visible on the left side. The toolbar at the bottom is also present. The window title "w1" and the standard Windows controls are consistent with the first screenshot.



Adding Tabs

To add a tab, you have to specify one more SCREEN section in your form file, just after the first SCREEN section. You can add *TITLE "Tab title"* structure after the SCREEN keyword to specify the names for the tabs (*Tab_1* and *Tab_2* in the pictures above). If you do not specify any names for your form pages, they will be named automatically as *Screen1*, *Screen2*, *Screen3*, etc.

Here is the general syntax for a form tab.

```
SCREEN [SIZE size BY characters][KEY key_name] TITLE "title"{
  ...
}
```

The KEY clause is optional and is used to associate the tab with a key. This key can then be used to switch to this tab at runtime.

The ATTRIBUTES section of a multi-tab form contains the attributes for all the fields located in all the tabs and is to be specified after the last SCREEN section. If you specify an ATTRIBUTES between two SCREEN sections, a compile-time error will occur.

The following source code will produce a form with two tabs, as it is shown in the pictures above (To test this sample, add a picture file named *Querix_logo.png* to the folder where the executable file is located):

```
# The form file name is "Myform"

DATABASE formonly

SCREEN KEY F2 TITLE "Tab_1"
{
\g-----\g

  [f001      ]

  [f002      ]

  [f003      ]

  [f004      ]

\g-----\g

}
```



```
SCREEN KEY F3 TITLE "Tab_2"
{
  \g-----\g

  [ f005      ]      [pic                  ]
  [ f006      ]
  [ f007      ]

  \g-----\g
}
```

ATTRIBUTES

```
f001 = formonly.f001 TYPE CHAR;
f002 = formonly.f002 TYPE CHAR;
f003 = formonly.f003 TYPE CHAR;
f004 = formonly.f004 TYPE CHAR;
f005 = formonly.f005 TYPE CHAR;
f006 = formonly.f006 TYPE CHAR;
f007 = formonly.f007 TYPE CHAR;
pic = formonly.pic,
      config = "querix_logo.png",
      widget = "image";
```

INSTRUCTIONS

```
DELIMITERS "[ ]"
```



If you make form tabs of different sizes, the form will get the size of the biggest one, and the smaller tabs will be displayed to the top left corner of the space allocated for the form.

If you display a multi-tab form in character mode, you won't be able to see the names of the tabs, but the tabs will be displayed to the screen when activated. This is how our form looks in character mode:

A black rectangular window representing a character-mode display. Inside, there are four small square boxes arranged vertically, each containing a pair of brackets: [] on the first line, [] on the second, [] on the third, and [] on the fourth. A horizontal line is positioned above the first bracket and another below the last bracket.

- Tab_1

A black rectangular window representing a character-mode display. Inside, there are two pairs of square boxes. The first pair is on the left, and the second pair is on the right, both containing brackets: [] on the first line and [] on the second. A horizontal line is positioned above the first bracket and another below the last bracket.

- Tab_2

Switching Between Form Tabs

The switching between the form tabs is performed automatically when the program performs input from the form fields that are located in different tabs. You can see the pages that are not active by clicking their tabs, however, it won't make them active.

For example, you can make the program perform input from the field named f001 and then - from the field named f005. In this case, the program will activate *Tab_1* and, when the user enters the value and presses the Accept or the Return keys, it will switch to the tab named *Tab_2*:

```
MAIN

DEFINE a,b CHAR (10)

OPEN WINDOW win1 at 2,2 WITH FORM "myform"
```



```
ATTRIBUTES (BORDER)
```

```
INPUT a, b FROM f001, f005
```

```
END MAIN
```

If you run this application, you will see how the second form tab becomes active after you enter a value to the field f001 and press RETURN or ACCEPT.

You can also switch the tabs using the key specified in the KEY clause. This key can be used in an ON KEY clause during the input which will switch the display to the associated tab. For example:

```
MAIN
```

```
DEFINE a,b CHAR (10)
```

```
OPEN WINDOW win1 at 2,2 WITH FORM "myform" ATTRIBUTES (BORDER)
```

```
INPUT a, b FROM f001, f005
```

```
    ON KEY (F2)
```

```
        NEXT FIELD f001
```

```
    ON KEY (F3)
```

```
        NEXT FIELD f005
```

```
END INPUT
```

```
END MAIN
```

In this case you can click on the tab heading with the mouse to bring the tab forward, or you can press keys F2 and F3 associated with the tabs. If the tabs are not associated with any keys (or with these particular keys), only pressing these keys or moving the input to the field in another tab will switch the tabs, but the user will not be able to toggle them using the mouse cursor.

Calendar Widget

The calendar widget is a widget displayed like a function field, with the button to the right which, when clicked, opens a calendar window to let the user select the necessary date graphically using their mouse. When the user selects a date, it is passed to the field. The date can also be inputted manually by means of the keyboard.

The calendar widget field displays the date in the format, which is set as the default one in your system.

The general syntax of the attribute is as follows:



```
field_tag = formonly.field_name,  
           widget = "calendar"
```

As you can see, the calendar widget unlike other widgets doesn't have the *config* attribute.

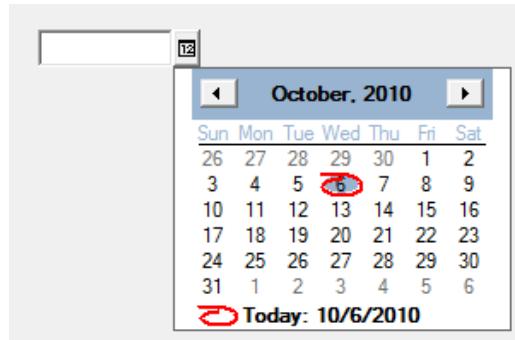
The following code creates a calendar widget:

```
# Form file "calend"  
  
DATABASE formonly  
  
SCREEN {  
  
    [ calen ]  
  
}  
  
ATTRIBUTES  
  
    calen = formonly.clndr,  
           widget = "calendar" ;  
  
INSTRUCTIONS  
    DELIMITERS " | "  
  
## Executable file  
MAIN  
  
DEFINE dt DATE  
  
OPEN FORM cal_f FROM "calend"  
DISPLAY FORM cal_f  
  
INPUT dt FROM clndr  
  
END MAIN
```

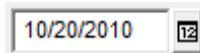
When the program is launched, the user will see an empty calendar field:



When they click on the button to the right, they will be allowed to choose the date:



After the user chooses the month and the day, the calendar window will close and the chosen date will be displayed to the field:



Hotlink Widget

The hotlink widget creates a label that, when the user clicks it, opens a web browser with a specified URL or a file path, or opens a file explorer. The type of the widget, as well as the class of event (whether the web browser or file explorer is to be opened) are specified in the ATTRIBUTES section within the form file.

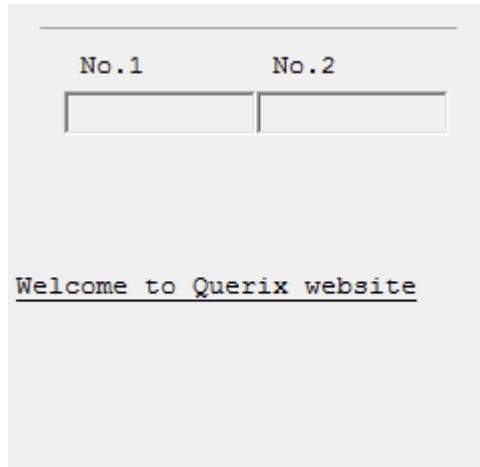
The attributes specification of the widget is as follows:

```
field_tag = database.field_name  
  
config = "URL or file path {Displayed label}"  
  
widget = "hotlink"  
  
class = "class"
```

The *class* attribute is not obligatory. If you omit it, the widget will launch an external browser. The same effect will be attained if you specify the *class* attribute as "default". You can also specify a key press or an action by specifying the *class* attribute as "key". The key events and actions will be discussed later in this manual.

The following lines can be specified for the field f003 if you want to make it a hotlink widget:

```
f003=formonly.mylink,  
  
config = "www.querix.com {Welcome to Querix website}",  
  
widget = "hotlink";
```



You can change the URL and the label of the hotlink using the DISPLAY TO statement, which is discussed later in this chapter.

To set a new label and a new URL for a hotlink, specify them in quotes just as you do it in the *config* attribute within the form file:

```
DISPLAY "www.google.com{Click to search in the Internet}" TO mylink
```

If you do not specify a new label, the one specified earlier will remain. If the application is run in the character mode, the hotlink field remains empty.

Example

This application illustrates different input modes and input control clauses. It also contains a form with form tabs and shows how the user can switch between them.

```
#####
# This example illustrates the usage of the INPUT statement
#####
DEFINE
    n_key      INTEGER,
    answer     CHAR,
    info       STRING,
    r_all_data RECORD
        id          INTEGER,
        first_name  CHAR(30),
        last_name   DATE,
        date_birth  INTEGER,
        phone       CHAR(13),
        units_pd    DECIMAL(8,2),
        price_pu   MONEY(16,2),
        id_card    CHAR(10),
        date_issue  DATE,
        date_expire VARCHAR(80),
        address    STRING,

```



```
        city          CHAR( 20 ),
        zip_code      CHAR( 6 ),
        country_name  CHAR( 20 ),
        add_data      CHAR(512),
        start_date    DATE,
        start_time    DATETIME HOUR TO SECOND,
        end_date      DATE,
        end_time      DATETIME HOUR TO SECOND,
        accept_date   DATE
    END RECORD,

r_person   RECORD
    first_name,
    last_name   CHAR( 30 ),
    date_birth  DATE,
    age         INTEGER,
    phone       CHAR(14),
    address     VARCHAR(80),
    city        CHAR( 20 ),
    zip_code    CHAR( 6 ),
    country_name CHAR( 20 )
END RECORD,

r_full_address RECORD
    address      VARCHAR(80),
    city         CHAR( 20 ),
    zip_code    CHAR( 6 ),
    country_name CHAR( 20 )
END RECORD,

MAIN

# This application uses widgets non-displayable
# in character mode, so we need to perform a check
IF fgl_fglgui()= 0 THEN
    CALL fgl_winmessage( "Wrong mode",
        "Run this application in the GUI mode.", "info" )
    EXIT PROGRAM
END IF

OPTIONS
ACCEPT KEY F3 -- F3 is assigned as the Accept key for the whole program

# This menu allows the user to browse between different types of input
# each menu option refers to a function declared below in this module
MENU "Input"

COMMAND "Normal INPUT",
    "Perform input in all form fields available using INPUT...FROM"
    CALL all_input_fromonly()

COMMAND "THRU keyword",
    "Perform input in some form fields specified with the THRU keyword"
    CALL input_thru()
```



```
COMMAND "WITHOUT DEFAULTS",
    "Perform input without the default values being displayed to the fields"
    CALL without_defaults()

COMMAND "Form Tabs",
    "Open a form with two tabs available for input"
    CALL tabs()

COMMAND "EXIT",
    "Exit the program"
    EXIT PROGRAM

END MENU

END MAIN

#####
# This function enables the buttons on the form
#####
FUNCTION buttons()
    DISPLAY "SAVE" TO save_but
    DISPLAY "QUIT" TO quit_but
    DISPLAY "!" TO save_but
    DISPLAY "!" TO quit_but
END FUNCTION

#####
# This function enables user to perform the input in any form field.
# It also illustrates input control clauses and some built-in functions
#####
FUNCTION all_input_fromonly()

OPEN WINDOW w_app_form_2 AT 1,1 WITH FORM "app_form_2"
ATTRIBUTE (FORM LINE 2, COMMENT LINE LAST)
DISPLAY "The INPUT...FROM operator is used with all the form fields:"
TO label_main

# When the user presses the Accept key (F3) or when he leaves the last input
# field (accept_date) using Enter, Down or Right key,
# the INPUT statement is still in effect, because of the AFTER INPUT clause
INPUT r_all_data.* FROM all_fields.*

BEFORE INPUT -- the statements of this block are executed before
-- the form is activated

LET info = "The statements within the BEFORE INPUT",
    "\n clause are executed before the data are entered",
    "\n into the form"
CALL fgl_winmessage("BEFORE INPUT", info, "info")

# We activate buttons and display information to the labels
CALL buttons()

# Initialize some members of record r_all_data before the input begins
# The commented statements are included to illustrate all the members of
```



```

# the program record. However, they will not be initialized, while
# commented. These values will be displayed to the fields instead of the
# default values

LET r_all_data.id = 1
--   LET r_all_data.first_name = "John"
--   LET r_all_data.last_name = "Smith"
--   LET r_all_data.date_birth = "03/08/1965"
--   LET r_all_data.age = YEAR(TODAY) -
--                                     YEAR(r_all_data.date_birth)

LET r_all_data.phone = "066-385-03-22"
LET r_all_data.units_pd = 5.5
LET r_all_data.price_pu = 9.00
LET r_all_data.id_card = "2397213D"
LET r_all_data.date_issue = NULL
LET r_all_data.date_expire = NULL
LET r_all_data.address = "102, Everton Rd."
--   LET r_all_data.city = "Liverpool"
--   LET r_all_data.zip_code = "123456"
LET r_all_data.country_name = "United Kingdom"
LET r_all_data.add_data = ""

"Here the additional comments are displayed to a multisegment field. ",
"This field can comprise several lines of text either displayed to it",
"or entered by a user. In a form file a multi-segment field looks like",
"several fields with the same tag placed one under another. ",
"The WORDWRAP attribute is required to create a multi segment field."
--   LET r_all_data.start_date = TODAY
--   LET r_all_data.start_time = CURRENT HOUR TO SECOND
LET r_all_data.end_date = "12/31/9999"
LET r_all_data.end_time = "23:59:59"
--   LET r_all_data.accept_date = TODAY
DISPLAY BY NAME r_all_data.*

BEFORE FIELD age -- this clause is activated when the cursor enters the
-- formonly.age field

# We use this clause to prevent this field from being inputted,
# while the value of this field is calculated by the program.
# If you press RIGHT, DOWN or RETURN to move the cursor out of "date_birth"
# field, it will be moved to "phone" field and not to "age" field
IF FGL_LASTKEY() = FGL_KEYVAL("right") OR
   FGL_LASTKEY() = FGL_KEYVAL("down") OR
   FGL_LASTKEY() = FGL_KEYVAL("return")
THEN NEXT FIELD NEXT
END IF
# If you press LEFT or UP key to move the cursor out of "phone" field, it
# will be moved to the previous field "date_birth" and not to "age" field
IF FGL_LASTKEY() = FGL_KEYVAL("Left") OR
   FGL_LASTKEY() = FGL_KEYVAL("Up")
THEN NEXT FIELD PREVIOUS
END IF

ON KEY (F10)      -- this key should be pressed, if you want to complete

```



```

-- the input. Otherwise the input will continue and you
-- will be returned to the first field after
-- leaving the last one due to the AFTER FIELD
clause

# The user will be asked to confirm their decision to quit the input,
# If they press "No", the cursor will be moved to the first form field
# and the input will continue. Otherwise, the menu will be redisplayed
LET answer = fgl_winquestion("QUIT?", "Do you really want to
quit?", "No", "Yes|No", "question", 1)
    # We use value "Y" because answer variable has size 1
    # so only the first character of the returned value
    # is recorder into it
    IF answer MATCHES "Y" THEN
        EXIT INPUT
    ELSE CONTINUE INPUT
    END IF

AFTER FIELD first_name -- this clause is executed before the cursor leaves
-- "first_name" field

# If you press UP or LEFT with the cursor at the beginning of "first_name"
# field, the cursor will be moved to the last form field "accept_date" which
# will loop the movement of the cursor through the form

IF FGL_LASTKEY() = FGL_KEYVAL("UP") OR
    FGL_LASTKEY() = FGL_KEYVAL("LEFT")
THEN NEXT FIELD accept_date
END IF
IF FGL_LASTKEY() = FGL_KEYVAL("DOWN") OR
    FGL_LASTKEY() = FGL_KEYVAL("RIGHT")OR
    FGL_LASTKEY() = FGL_KEYVAL("RETURN")
THEN NEXT FIELD NEXT
END IF

AFTER FIELD date_birth -- before the cursor leaves this field, the entered
-- data is verified to be sure the default date is
-- not left in the field
IF r_all_data.date_birth IS NULL OR
    r_all_data.date_birth <= "12/31/1899"
THEN ERROR "Enter a date later than 12/31/1899"
NEXT FIELD date_birth
END IF
LET r_all_data.age = YEAR(TODAY) - YEAR(r_all_data.date_birth)
DISPLAY BY NAME r_all_data.age

AFTER FIELD accept_date -- this clause is executed when the cursor is
-- about to leave the last form field

# If you press DOWN, RIGHT, or RETURN key when in this field, the cursor
# will be moved to the first form field looping the cursor movement
IF FGL_LASTKEY() = FGL_KEYVAL("Down") OR
    FGL_LASTKEY() = FGL_KEYVAL("Right")OR
    FGL_LASTKEY() = FGL_KEYVAL("Return")
THEN NEXT FIELD first_name

```



```

        END IF
        IF FGL_LASTKEY( ) = FGL_KEYVAL( "Left" ) OR
           FGL_LASTKEY( ) = FGL_KEYVAL( "Up" )
        THEN NEXT FIELD PREVIOUS
        END IF

        AFTER INPUT -- this clause is executed then the Accept key (F3) is pressed
                      -- it prevents the input from terminating and tests some
                      -- entered values you will be returned to this field, which
                      -- value does not meet the requirements
        IF r_all_data.first_name IS NULL OR
           LENGTH(r_all_data.first_name) = 0 -- the length() function tests
                                             -- the number of characters
                                             -- contained in the value
        THEN ERROR " Enter a value in the field " -- the message will appear at
                                             -- the bottom of the screen
                NEXT FIELD first_name
        END IF
        IF r_all_data.last_name IS NULL OR
           LENGTH(r_all_data.last_name) = 0
        THEN ERROR " Enter a value in the field "
                NEXT FIELD last_name
        END IF
        IF r_all_data.date_birth IS NULL OR
           r_all_data.date_birth <= "12/31/1899" -- prevents the default value
                                             -- from being retrieved from
                                             -- the input
        THEN ERROR " Enter a date later than 12/31/1899"
                NEXT FIELD date_birth
        END IF
        IF r_all_data.date_issue IS NULL OR
           r_all_data.date_issue <= "12/31/1899"
        THEN ERROR " Enter a date later than 12/31/1899"
                NEXT FIELD date_issue
        END IF
        IF r_all_data.date_expire IS NULL OR
           r_all_data.date_expire <= "12/31/1899"
        THEN ERROR " Enter a date later than 12/31/1899"
                NEXT FIELD date_expire
        END IF
        IF r_all_data.date_issue >= r_all_data.date_expire
        THEN ERROR " The expire date must not be earlier than the date of issue"
                NEXT FIELD date_issue
        END IF
        CALL fgl_winmessage ( "DATA SAVED", "The data was saved", "info" )

        CONTINUE INPUT -- after all the tests are done, the input is still
                      -- active it will be terminated only if the
                      -- ON KEY (F10) clause is activated

        END INPUT
        CLOSE WINDOW w_app_form_2
        CLEAR SCREEN
END FUNCTION

```



```
#####
# This function allows the user to input values into a restricted
# subset of fields and not to all the form fields
#####
FUNCTION input_thru()

OPEN WINDOW w_app_form_2 AT 1,1 WITH FORM "app_form_2"
    ATTRIBUTE (FORM LINE 2)

    # Again we need to activate buttons and labels
    CALL buttons()

DISPLAY "The INPUT...FROM statement with THRU option:" TO label_main

    # We assign some values to the variables associated with the input
    # They will be displayed to fields as the default values
    LET r_person.last_name = "Smith"
    LET r_person.date_birth = "01/01/2000"
    LET r_person.age = 0
    LET r_person.phone = "123-456-78-90"

    # Without the WITHOUT DEFAULTS keywords the current values of the variables
    # associated with the input(r_person.last_name,r_person.age and
    # r_person.phone) are not displayed to the fields of the form, while the
    # default values of the fields are in effect

INPUT r_person.last_name THRU r_person.phone
    FROM s_person.last_name,s_person.date_birth,s_person.age,s_person.phone
    ATTRIBUTE (CYAN, REVERSE)

    BEFORE INPUT
        LET info = "The values of the variables will not",
            "\n be displayed to the fields, because",
            "\n the WITHOUT DEFAULTS keywords",
            "\n are absent."
        CALL fgl_winmessage("Field values", info, "info")

    # When you press the Accept key (F3) or leave the last form field using
    # the ENTER, DOWN or RIGHT key, the input is terminated, because the
    # AFTER FIELD and AFTER INPUT clauses are not used to loop the cursor
    # movement
    AFTER FIELD date_birth
        IF r_person.date_birth IS NULL OR
            r_person.date_birth <= "12/31/1899"
        THEN ERROR " Enter a date later than 12/31/1899"
            NEXT FIELD date_birth
        END IF
        LET r_person.age = YEAR(TODAY) - YEAR(r_person.date_birth)
        DISPLAY BY NAME r_person.age -- we calculate and display the age value
                                      -- based on the entered birth date

    ON KEY (F10)
        LET answer = fgl_winquestion("QUIT?",
            "Do you really want to quit?", "No", "Yes|No", "question", 1)
```



```
IF answer = "Y" THEN
    EXIT INPUT
ELSE
    CONTINUE INPUT
END IF

END INPUT
CLOSE WINDOW w_app_form_2
CLEAR SCREEN
END FUNCTION

#####
# This function allows the user to enter values into a restricted
# set of fields, while the default values are not displayed
# to these fields
#####
FUNCTION without_defaults()

OPEN WINDOW w_app_form_2 AT 1,1 WITH FORM "app_form_2"
    ATTRIBUTE (FORM LINE 2)

DISPLAY "INPUT...FROM with THRU option and WITHOUT DEFAULTS keywords:"
    TO label_main

# With the WITHOUT DEFAULTS option in effect the values of variables
# r_person.last_name, r_person.age and r_person.phone changed above will be
# displayed to the form fields instead of the default values

INPUT r_person.first_name THRU r_person.zip_code WITHOUT DEFAULTS
    FROM s_person.first_name, s_person.last_name, s_person.date_birth,
        s_person.age, s_person.phone, s_person.address,
        s_person.city, s_person.zip_code
    ATTRIBUTE (CYAN, REVERSE)
BEFORE INPUT
LET info = "If you have used this menu option before",
    "\n the values you entered will be displayed",
    "\n in the input fields due to the presence",
    "\n of the WITHOUT DEFAULTS keywords."
CALL fgl_winmessage("Default Values",info, "info")
CALL buttons()

# When you press the Accept key (F3) or leave the last form field using
# the ENTER, DOWN or RIGHT key, the input is terminated, because the
# AFTER FIELD and AFTER INPUT clauses are not used to loop the cursor
# movement
AFTER FIELD date_birth
    IF r_person.date_birth IS NULL OR
        r_person.date_birth <= "12/31/1899"
    THEN ERROR " Enter a date later than 12/31/1899"
        NEXT FIELD date_birth
    END IF
    LET r_person.age = YEAR(TODAY) - YEAR(r_person.date_birth)
    DISPLAY BY NAME r_person.age
```



```
ON KEY (F10)
LET answer = fgl_winquestion("QUIT?",
    "Do you really want to quit?", "No", "Yes|No", "question", 1)
IF answer MATCHES "Y" THEN
    EXIT INPUT
ELSE CONTINUE INPUT
END IF

END INPUT
CLOSE WINDOW w_app_form_2
CLEAR SCREEN
CLEAR SCREEN
END FUNCTION

#####
# This function opens a form with two tabs available for input
#####
FUNCTION tabs()

DEFINE browser_url,
    hotlink_url_1,
    hotlink_url_2,
    browser,
    hotlink1,
    hotlink2 VARCHAR(25)

# We display a form with two tabs
OPEN WINDOW brows_form AT 1,1 WITH FORM "form_tabs"
DISPLAY "!" TO switch
DISPLAY "!" TO return_but
DISPLAY "!" TO qt

# First the input is performed only into the
# fields of the first tab
INPUT browser_url, hotlink_url_1, hotlink_url_2
FROM browser_url, hl_url_1, hl_url_2

# This key is associated with the "Switch tabs" button
# and with the header of the second tab

ON KEY (F6)
    # The nested input into the fields of the second tab
    # makes this tab active and brings it forward
    INPUT browser, hotlink1, hotlink2
    FROM browser, hotlink_1, hotlink_2

    # We send all the URSSs specified in the first tab
    # to the fields of the second tab so they can be used
BEFORE INPUT
    DISPLAY browser_url TO browser
    DISPLAY hotlink_url_1 TO hotlink_1
    DISPLAY hotlink_url_2 TO hotlink_2

    # This action is associated with the button "RETURN"
    # and with the header of the first tab.
```



```

# It finishes the nested input and the program control
# is returned to the input in the first tab.
ON KEY(F5)
    EXIT INPUT

END INPUT

# F10 is the only way to exit input,
# because they have the CONTINUE INPUT keywords
# in the AFTER INPUT clause
ON KEY (F10)
    LET answer = fgl_winquestion("QUIT?",
        "Do you really want to quit?", "No", "Yes|No",
        "question", 1)
    IF answer MATCHES "Y" THEN
        EXIT INPUT
    ELSE
        CONTINUE INPUT
    END IF

# We loop the input so that the user could enter the new URLs
# and check the results continuously
AFTER INPUT
    CONTINUE INPUT

END INPUT

CLOSE WINDOW brows_form
CLEAR SCREEN
END FUNCTION

```

Form Files

The first form is called "app_form_2.per"

```

DATABASE formonly
SCREEN SIZE 28 BY 80
{
    [label_main
\g-----\g[app_form
        ID:[f00      ]                                ]
        First Name:[f01                            ]
        Last Name:[f02                            ]
        Birth Date:[f05      ]    Age:[f6     ]      ]
        Phone:[f07      ]                                ]
\g-----\g[id_card_lab]\g-----\g
        ID Number:[f10      ]    Issued on: [f11      ]    Expires on: [f12      ]
\g-----\g[address_lab]\g-----\g
        Address:[f13      ]                                ]
        City:[f14      ]    Zip:[f15      ]    Country:[f17      ]
\g-----\g[add_lab
        [f18      ]                                ]
        [f18      ]                                ]

```



```
[f18] ]\g-----\g[reg_lab      ]\g-----\g
\g-----\g[Registered:[f19      ] at [f20      ] Dismissed: [f21      ] at [f22      ]
Accepted:[f23      ] ]\g-----\g
} [f26      ] [f27      ] \g-----\g

ATTRIBUTES
# First, we define the dynamic labels that serve to name the form divisions.
# They will receive values at runtime
label_main = formonly.label_main, widget = "label",
            config = "Main descriptive label", COLOR = GREEN;
app_form = formonly.app_form, widget="label",
            config="Application Form",COLOR=BLUE, CENTER;
id_card_lab = formonly.id_card_lab, widget="label", config="ID Card",
              CENTER, COLOR=BLUE;
address_lab = formonly.address_lab, widget="label", config="Address",
              CENTER, COLOR=BLUE;
add_lab = formonly.add_lab, widget="label", config="Additional Data",
              CENTER, COLOR=BLUE;
reg_lab = formonly.reg_lab, widget="label", config="Registration",
              CENTER, COLOR=BLUE;

-- The COMMENTS attribute contains the message which is displayed when the
-- cursor is in the field
f00 = formonly.id TYPE INTEGER,COLOR = YELLOW REVERSE ,NOENTRY;
                  -- this field cannot accept values because of the
                  -- NOENTRY attribute
f01 = formonly.first_name TYPE CHAR,COMMENTS=" Enter the first name";
f02 = formonly.last_name TYPE CHAR,COMMENTS=" Enter the last name";
f05 = formonly.date_birth TYPE DATE, FORMAT="mm.dd.yyyy", PICTURE="#"##.#####",
      CENTURY = "R", COMMENTS=" Enter the date birth";
                  -- the FORMAT and PICTURE
                  -- attributes are used to adjust
                  -- the appearance of the field the CENTURY attribute
                  -- will take effect, if you enter only 2 digits for the year

f6 = formonly.age TYPE INTEGER;
f07 = formonly.phone, PICTURE = "###-###-##-##",
      COMMENTS=" Enter the phone number";           -- The PICTURE attribute
displayed
                                         -- hyphens to the field
f08 = formonly.units_pd TYPE DECIMAL,COMMENTS=" Enter the units per day";
f09 = formonly.price_pu TYPE DECIMAL,COMMENTS=" Enter the price per unit";

f10 = formonly.id_card TYPE CHAR, COMMENTS=" Enter the id card number";

# The following two fields are calendar widgets. Now you can either enter a date
# or select it by clicking the button to the right of the field
f11 = formonly.date_issue TYPE DATE, widget = "calendar", FORMAT="mm/dd/yyyy",
      PICTURE="#"##.#####", COMMENTS=" Enter the date the ID was issued";
f12 = formonly.date_expire TYPE DATE, widget = "calendar", FORMAT="mm/dd/yyyy",
      PICTURE="#"##.#####", COMMENTS=" Enter the date when ID expires";
```



```

f13 = formonly.address TYPE VARCHAR,
      COMMENTS=" Enter the street, house number, apartment, etc.";
f14 = formonly.city TYPE CHAR;
f15 = formonly.zip_code TYPE CHAR,
      COMMENTS=" Enter the zip code for this address";
f17 = formonly.country_name,NOENTRY; -- This field cannot accept values, as all
-- the cities belong to the country already
-- specified

f18 = formonly.add_data TYPE CHAR,WORDWRAP,
      COMMENTS=" Enter the additional data";
f19 = formonly.start_date TYPE DATE,DEFAULT = TODAY,NOENTRY;
-- The DEFAULT attribute assigns the default date and NOENTRY
-- prevents it from being changed

f20 = formonly.start_time TYPE DATETIME HOUR TO SECOND,
      DEFAULT = CURRENT, NOENTRY;
f21 = formonly.end_date TYPE DATE, widget ="calendar",
      COMMENTS=" Enter the date of dismissing";
f22 = formonly.end_time TYPE DATETIME HOUR TO SECOND,
      COMMENTS=" Enter the time of dismissing";
f23 = formonly.accept_date TYPE DATE, widget ="calendar",
      DEFAULT = TODAY, COMMENTS=" Enter the date of accepting";

# This button functions as the ACCEPT key. After this button is pressed
# the input values will be saved to the variables and the input will be
# continued. There is no ON KEY clause for this key, because
# F3 was made the default ACCEPT key
f26 = formonly.save_but, widget="button", config ="F3",
      COLOR = GREEN BOLD, CENTER;
f27 = formonly.quit_but, widget="button", config="F10", COLOR = RED BOLD,
      CENTER;

```

INSTRUCTIONS

```

DELIMITERS "[ ]"
SCREEN RECORD s_person (formonly.first_name, formonly.last_name,
                        date_birth, age, phone, address, city, zip_code,
                        country_name)
SCREEN RECORD s_full_address (address THRU country_name)
SCREEN RECORD all_fields (formonly.id THRU formonly.accept_date)

```

The second form is called "form_tabs.per". It is a multi-tab form, so it has two SCREEN sections. However, there is only one ATTRIBUTES section used for the both screens.

DATABASE **formonly**

```

-- This form has two pages each with a browser widget on it. When you open the
-- form, there will be two tabs at the top used to switch between the pages
SCREEN KEY F5 TITLE "Input Data"

```

```
{

```

```
[main_screen1
```

```
]
```





```
-- clicking a hotlink widget will open the URL in the default browser
hotlink_1 = formonly.hotlink_1, widget = "hotlink",
            config ="blank {First alternative link}", COLOR = BLUE UNDERLINE;
hotlink_2 = formonly.hotlink_2, widget = "hotlink",
            config ="blank {Second alternative link}", COLOR = BLUE UNDERLINE;
```

INSTRUCTIONS

DELIMITERS "[]"



Ways of Producing Output

4GL offers some other ways to produce output except for the DISPLAY statement and other statements you already know. In this chapter we will discuss other ways of displaying different types of messages to the screen or a window. We will also explain how the contents of a text file, an image or another binary object can be displayed or edited within a 4GL application.

There is one more way of performing the output which is producing reports. However, this method will be discussed at the end of this manual. It is mainly used to process the data received from database queries, but it can be used to process any kind of data. Thus, this method will be introduced after you master all the ways of data manipulation in 4GL.

Errors and Messages

Throughout the execution of a 4GL application you may need to display some standard messages to a user to show that an error has occurred, to warn him about the possible result of his actions, etc. It is not always convenient to use the DISPLAY statement for these purposes, while you need to be careful when you select the position of the displayed line in order not to interfere with any other information displayed.

It is convenient to use reserved lines for displaying such information. Their position can be changed, thus you can select such a position where they will not interfere with other displayed information. 4GL has two statements which display their messages on the reserved lines: these are MESSAGE and ERROR statements.

The MESSAGE Statement

The MESSAGE statement displays a character string to the reserved message line. The syntax of this statement is as follows:

```
MESSAGE displayed_value [ATTRIBUTE clause]
```

The displayed value can be represented by a variable, by a quoted character string, or by their combination. In case if you combine several variables or variables and character strings, all elements must be separated by commas. The displayed value can occupy only one line of the screen or a window. If the length of the message text exceeds the width of the screen or 4GL window, the text is truncated to fit.

The message of the MESSAGE statement is displayed in the current window on the reserved message line which is the second line of the window or screen by default. The example below will display the string "Today is <current date>" to the second line of the 4GL screen:

```
MAIN
DEFINE my_date DATE
LET my_date = TODAY
MESSAGE "Today is ", my_date
END MAIN
```

In the displayed value you can use all the operators normally used with character strings, such as [], ||, CLIPPED, etc. For more details about these operators see "[The Notion of Variables](#)".

The message statement remains on the screen until another message is displayed. To clear the message from the screen you can display an empty string to the message line:



```
MESSAGE " "
```

The ATTRIBUTE Clause

By default the message is displayed with the NORMAL attribute. You can add the optional ATTRIBUTE clause to change the display mode. The ATTRIBUTE clause can accept all the standard display attributes of the DISPLAY statement (colour and intensity). However, the INVISIBLE attribute will be ignored, if you apply it to the MESSAGE statement.

```
MESSAGE "This is my message" ATTRIBUTE (CYAN, REVERSE)
```

The MESSAGE LINE

The message line is the line of the current window where the MESSAGE statement performs output. By default the message line is the second line of the 4GL screen or window. It may interfere with some other objects (such as MENU which takes the first and the second line). If some other object occupies the reserved line, the message is not displayed.

The position of the message line can be changed in two ways:

- By specifying the MESSAGE LINE attribute of the ATTRIBUTE clause of a window where you want to display the message:

```
OPEN WINDOW my_win AT 2,2 WITH 20 ROWS, 60 COLUMNS
ATTRIBUTE(BORDER, MESSAGE LINE LAST-3)
```

- By specifying the MESSAGE LINE in an OPTIONS statement:

```
OPTIONS MESSAGE LINE 20
```

The position of the MESSAGE LINE can be specified using the line number or the FIRST and LAST keywords in the same way as for the [PROMPT LINE](#) and MENU LINE.

The ERROR Statement

The ERROR statement displays an error message on the error line and rings the terminal bell. The syntax of this statement is similar to that of the MESSAGE statement:

```
ERROR displayed_value [ATTRIBUTE clause]
```

The displayed value is the message which is displayed by the ERROR statement on the error line. The requirements for the displayed value correspond to that of the displayed value of the MESSAGE statement described above.

Unlike the MESSAGE statement, the character string displayed by the ERROR statement disappears from the screen when the user presses the next key.

The ATTRIBUTE Clause

By default the error message is displayed with the REVERSE attribute. The ATTRIBUTE clause of the ERROR statement supports all standard display attributes. The INVISIBLE attribute is ignored, if it is combined with any other attribute. If the INVISIBLE attribute is the only attribute, the error message is displayed as NORMAL.



```
ERROR "This is an error message number ", err_num
ATTRIBUTE (RED, REVERSE)
```

The ERROR LINE

The line to which the ERROR statement produces output is called the ERROR LINE. By default, it is the last line of the 4GL Screen (and not of the current window). The position of the error line is relative to the screen and not to the current window, thus it can be changed only by means of the OPTIONS statement:

```
OPTIONS ERROR LINE LAST-3
```

The line number can be specified in the same way as for the [PROMPT statement](#).

Querix 4GL also supports a script option that allows you to hide the error line when the program is run in the GUI mode.

Functions Displaying Help

There are two ways of specifying the help messages and displaying them. One is the HELP clause of some statements (for example, in the PROMPT or the INPUT statements). These HELP causes are applied to the whole statement and the help messages specified in them are displayed whenever the user presses the Help key during the statement execution.

However, there is a way to specify help messages that will be displayed under special conditions (for example, when the cursor is positioned to the field, or when some condition in a conditional clause is satisfied, or when an error occurs).

The showhelp() Function

The *showhelp()* function is used to display a help message. The argument of the function is a SMALLINT value which indicates the number of the message in the current help file that is to be displayed. This function can invoke a help message regardless of whether a user pressed the help key or not unlike the HELP clause in many statements which requires the help key to be explicitly pressed.

The syntax of the function invocation is:

```
CALL showhelp (number)
```

The help file which contains the specified message should be created and compiled as it was described earlier when discussing the [PROMPT statement](#). You shouldn't also forget to specify the help file in the HELP FILE clause of the OPTIONS statement.

The Help Menu

When the program executes the *showhelp()* function, it opens the help window. The window contains the text of the help message and a ring menu on the top of the window. This menu is called the *Help menu*.

If the help message needs several pages to be displayed, the *showhelp()* function displays the first one, and the rest of the text can be displayed by means of the *Screen* option of the Help menu. The *Resume* option makes the program close the window and returns the control to the 4GL screen:



```
C:\WINDOWS\system32\cmd.exe
Help: Next Page | Previous Page | End
Display the previous help page

Main article: Classical order

Church of San Prospero, Reggio Emilia, ItalyThe Roman author Vitruvius, relying
on the writings (now lost) of Greek authors, tells us that the ancient Greeks
believed that their Doric order developed from techniques for building in wood i
n which the earlier smoothed tree trunk was replaced by a stone cylinder.

Doric order
Main article: Doric order
The Doric order is the oldest and simplest of the classical orders.
It is composed of a vertical cylinder that is wider at the bottom. It generally
has neither a base nor a detailed capital. It is instead often topped with an
inverted frustum of a shallow cone or a cylindrical band of carvings.
It is often referred to as the masculine order because it is represented in the
bottom level of the Colosseum and the Parthenon, and was therefore considered to
be able to hold more weight. The height-to-thickness ratio is about 8:1.
The shaft of a Doric Column is always fluted.

The Greek Doric, developed in the western Dorian region of Greece, is the heaviest
and most massive of the orders. It rises from the stylobate without any base;
it is from four to six times as tall as its diameter; it has twenty broad flutes;
the capital consists simply of a banded necking swelling out into a smooth echinus,
which carries a flat square abacus; the Doric entablature is also the heaviest,
being about one-fourth the height column. The Greek Doric order was not used
after c. 100 B.C. until its rediscovery in the mid-eighteenth century.

Tuscan order
Main article: Tuscan order
The Tuscan order, also known as Roman Doric, is also a simple design, the base
and capital both being series of cylindrical disks of alternating diameter.
The shaft is almost never fluted. The proportions vary, but are generally similar
You have reached the end of this help text.
```

The *showhelp()* Function Application

When the *showhelp()* function is used in such statements as INPUT, INPUT ARRAY, PROMPT or within the COMMAND clause of the MENU statement, its effect is very similar to the effect of the Help key. However, unlike with the Help key which displays one and the same message throughout the whole program, the *showhelp()* function provides the possibility to display different messages for different occasions.

In the following example, the program displays different help messages depending on the current field:

```
INPUT ARRAY scr_arr FROM scr_rec.*

ON KEY (CONTROL-B)

CASE

    WHEN INFIELD(f_name)
        CALL showhelp(101)

    WHEN INFIELD(l_name)
        CALL showhelp(102)

    WHEN INFIELD(dt)
        CALL showhelp(103)

END CASE
```



END INPUT

Don't forget to specify the help file within the OPTIONS statement before you use the *showhelp()* function.

If your program has the following HELP file:

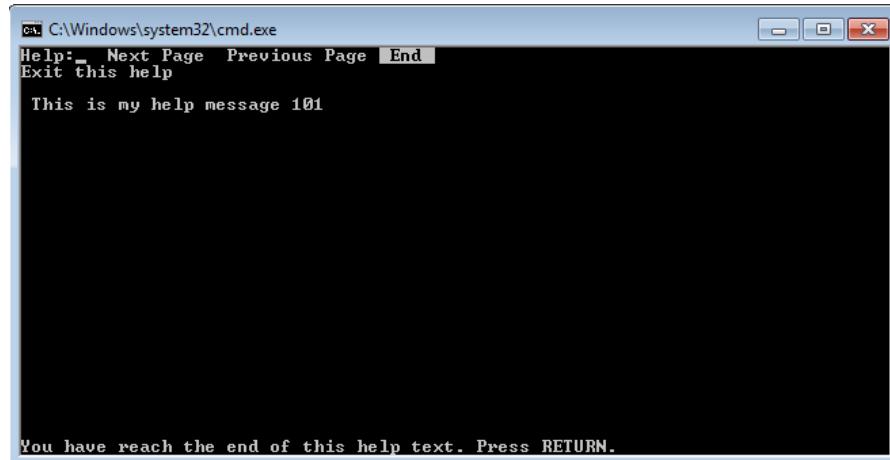
```
##Help file "Help.msg"
.100
This is my help message 100

.101
This is my help message 101

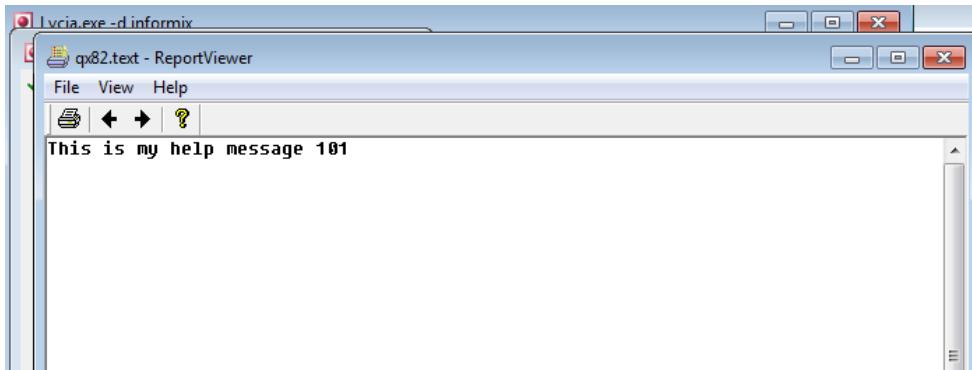
.102
This is my help message 102

.103
This is my help message 103
```

and the cursor is positioned to the field *f_name*, you will see the help message as follows:



- Character mode



- GUI mode (Phoenix)



The fgl_gethelp() Function

The *fgl_gethelp()* function is used to return a character string which contains the text of the SQL or 4GL error message. The number of this error is specified as the function argument:

```
Fgl_gethelp (integer value)
```

The integer value here can be represented by an integer number or by a variable containing such an integer. The argument of the *fgl_gethelp()* function is usually the value of the *status* variable which indicates the number of the error that has occurred, if any. In the following example, the function returns the error message and assigns it to a character variable *error_text*:

```
MAIN

DEFINE err INT,
      error_text VARCHAR (100)

LET err = -1142

IF err < 0 THEN

    LET error_text = fgl_gethelp(err)

    ERROR error_text -- Displays "Window too small to display form"
END IF

MESSAGE "Press any key to quit"

CALL fgl_getkey()

END MAIN
```

During the program execution, the variable *error_text* gets the value which corresponds to the error description, and this description is displayed to the error line as soon as the IF condition is satisfied.

Large Data Types

We have discussed almost all data types available in 4GL in the previous chapters. The only two data types left are the so called large data types. These are:

- The BYTE data type - variables of this data type store any binaries such as images, executable files, etc.
- The TEXT data type - variables of this data type are used to store printable characters.

The TEXT Data Type

The TEXT data type is used to store character data in ASCII strings. A TEXT variable can include all printable characters and the following white space characters:

- TAB (CONTROL-I)



- NEWLINE (CONTROL-J)
- FORMFEED (CONTROL-L)

Other non-printable characters might cause 4GL to process a TEXT variable incorrectly.

A TEXT variable can store values up to 2^{31} bytes long, but the size can be limited by the storage capacity of your system. A TEXT variable requires the LOCATE statement to be executed before it can be used. The LOCATE statement specifies where the value of the variable must be stored. It is described later in this chapter.

Restrictions

There are some restrictions which concern the usage of the TEXT variables in 4GL applications.

- A TEXT variable can be displayed by the DISPLAY AT statements, but the length of the displayed value will be limited by a single window line; it can be also displayed more efficiently by the DISPLAY TO statement to a form field or by the DISPLAY statement (without any additional clauses) to the console window in the GUI mode.
- You cannot use the LET statement to assign the following values: NULL value, values of CHAR, VARCHAR, STRING and TEXT data types.
- If you display a TEXT variable to a form field, only the characters which fit the field length will be displayed, the rest will be truncated.
- To display long TEXT values or to edit them, you should use the PROGRAM attribute for a field of the TEXT data type. The PROGRAM attribute is introduced later in this chapter.
- A text variable can be displayed to a multi-segment field, but it cannot be edited in such a way.
- A field used to display a TEXT variable can accept only COLOUR, WORDWRAP and PROGRAM attributes.
- In the CALL and RETURN statements TEXT variables are passed rather by reference than by value. When you change the value of a TEXT variable in a function, this change is also visible for the calling routine without any additional values transfer.

The BYTE Data Type

The BYTE data type stores any kind of binary data in an unstructured byte stream. A BYTE variable can store program load modules, audio records, images and anything else that can be stored in digital format. A BYTE variable can store values up to 2^{31} bytes long, but the size can be limited by the storage capacity of your system.

A BYTE variable requires the LOCATE statement to be executed before it can be used. The LOCATE statement specifies where the value of the variable must be stored. It is described later in this chapter.

Restrictions

Like TEXT variables, BYTE variables also have some restrictions:

- You cannot use LET statement to assign any values except NULL to a BYTE variable.
- In CALL and RETURN statements BYTE variables are passed rather by reference than by value, thus they cannot be specified as function arguments or returned values. When you change the value of a BYTE variable in a function, this change is also visible for the calling routine without any additional values transfer.



- None of the DISPLAY statements type can display BYTE variables. If a BYTE variable is displayed to a form field, the field will display <BYTE value> string instead of the value of a variable.
- To display BYTE values or to edit them, you should use the PROGRAM attribute for such field. Then you should specify the field and the variable in an INPUT statement. For details see the PROGRAM attribute below.
- A field of the BYTE data type can accept only COLOUR and PROGRAM attributes.

Initializing Large Variables

TEXT and BYTE variables are declared in the same way as any other variables, e.g.:

```
DEFINE
    t_var TEXT,
    b_var BYTE
```

However, you cannot yet use these variables even if they have been declared. To be able to use them you need to specify where their values should be stored with the help of the LOCATE statement.

The LOCATE statement specifies where to store large values, it has the following syntax:

```
LOCATE variable IN [MEMORY|FILE[file_name]]
```

Here “variable” stands for a variable of TEXT or BYTE data type. It can be a simple variable, a member of a record, if this member is of large data type or an array element of a large data type. The “file name” is the quoted name of a file (or a variable containing such name) which will be used for storing the value. The LOCATE statement must be used within the scope of reference of the variable you want to locate. It must also be used after the variable is declared, but before it is referenced by any other statements.

IN MEMORY

You can specify whether you want to store the value in the memory or in a file. The access to the value is faster, if it is stored in the memory. However, if the program exceeds the available memory, 4GL automatically stores part of the value in a file. The following piece of code stores the value of a BYTE variable in memory:

```
DEFINE b_var BYTE
LOCATE b_var IN MEMORY
```

The value stored in memory is discarded after the program is terminated.

IN FILE

You can locate a variable in a file. It can be either a named file which will remain on your system after the program execution is over, or a temporary file which will be deleted.

A Temporary File

To locate a variable in a temporary file, you should not specify the file name after the FILE keyword:

```
DEFINE t_var TEXT
LOCATE t_var IN FILE
```

A Named File

To locate a variable in a named file you should specify the file name with the path, if required:



```
DEFINE t_var TEXT
LOCATE t_var IN FILE "/source/my_file.txt"
```

You can use the LOCATE statement to specify a number of files at a time. This method is typically used to locate array elements. E.g.:

```
DEFINE
    file_name ARRAY[5] OF CHAR(20), -- variable used to
                                    -- store file names
    text_array ARRAY[5] of TEXT   -- the array of large data
                                    -- type
    i INTEGER

FOR i = 1 TO 5

#Here we specify the names of files which will be
"/source/textNo1", #"/source/textNo2", "/source/textNo3",
"/source/textNo4" and #"/source/textNo5"

LET file_name[i] = "/source/textNo", i, USING "<<&"

#Then we locate each array element in the corresponding file

LOCATE text_array[i] IN FILE file_name[i]

END FOR
```

You can use the LOCATE statement to specify another storage location for variables previously located in memory or in a temporary file. However, you cannot use this method for a variable located in a named file.

```
DEFINE
    var1 TEXT

LOCATE var1 IN MEMORY -- we can relocate it later
...
LOCATE var1 IN FILE    -- we can relocate it later
..
LOCATE var1 IN FILE "my_text.txt"
                    -- we cannot relocate it any more
```

Freeing Memory Allocated

If you do not need a TEXT or a BYTE variable any more, you can free the memory allocated to it. For this you can use two statements:

- FREE -this statement deletes the file, if a variable has been located in a file, and releases the memory, if it has been located IM MEMORY.
- LOCATE - this statement releases the memory and deletes the temporary file, but it does not delete named files.

After you release the storage, you cannot reference the TEXT or BYTE variable again without executing a new LOCATE statement to reinitialize it. If you have named the file for the TEXT or BYTE value, and you want to retain the file, do not use the FREE statement.



The `fgl_copyblob()` Function

In 4GL, all assignments of binary large objects (blobs) are performed by reference. It means that you cannot copy the contents of one blob to another using the same tools as when copying data from one variable to another one. For example, in the source code:

```
LOCATE ablob IN FILE "a.txt"
LOCATE bblob IN FILE "b.txt"
LET b = a
```

The LET statement is an invalid one and won't lead to the wanted result.

To copy a value from one blob into another, you should use the `fgl_copyblob()` function. Therefore, if you want to copy a value from "ablob" to "bblob", you should specify the function as:

```
LOCATE ablob IN FILE "a.txt"
LOCATE bblob IN FILE "b.txt"
Fgl_copyblob(bblob,ablob)
```

So, the `fgl_copyblob()` function needs two arguments, the first of which is the destination of the copy operation, and the second specifies the source from which the value is to be taken.

Editing Large Values

The PROGRAM attribute is used to specify an external application program to work with form fields of TEXT or BYTE data type. It allows to viewing and editing values of large data types. It has the following syntax:

```
PROGRAM = "command"
```

Here "command" is the quoted name of a command string (or the name of a shell script) that launches the application which can open and edit the value of a large variable. To activate the command you should press "!" at runtime, when the cursor is in the field with the PROGRAM attribute. If you do not use a database, this is the only way to assign and edit values of large data types.

The PROGRAM attribute should be added to a form field of the TEXT or BYTE data type in the ATTRIBUTES section of a form file. Then the large variable should be used in the INPUT statement referencing this field.

The following is a form file with a field of the BYTE data type. We presume that the value used would be an image, thus we specify the command launching the standard MS Paint application:

```
DATABASE formonly
SCREEN{
    [f001]
}
ATTRIBUTES
f001 = formonly.f001 TYPE BYTE, PROGRAM = "mspaint.exe";
END
```

The following code can be used with the above form file to illustrate the work of the PROGRAM attribute:

```
MAIN
DEFINE
    my_picture BYTE
```



```
LOCATE my_picture IN FILE "my_pic.jpg"
INPUT my_picture FROM f001
END MAIN
```

During the program execution press "!", when f001 field is active. This will open Paint for file edition.

Displaying and Editing Images

If a BYTE value is an image, it can be displayed to a field of the BYTE data type using the normal DISPLAY statement, if the PROGRAM attribute of this field is set to "bmp".

You can specify the "bmp" as the command for the PROGRAM attribute. In this case you will see the "Edit" button under the BYTE field and the value of the BYTE variable will be treated as an image and displayed to the field. Pressing the "Edit" button will be equivalent to pressing "!" key. The default image editor will be opened. If the value of the BYTE variable displayed to the field is not an image, the field will remain empty.

```
f001 = formonly.f001 TYPE BYTE, PROGRAM = "bmp";
```

Displaying and Editing Text

The TEXT values can be displayed to fields, but their length is restricted by the size of the field and they cannot be edited using a standard field. To display a TEXT variable into a field without any truncating and to be able to edit it freely, add the PROGRAM attribute with the "txt" value to the TEXT field. This will create an area with scroll bars if necessary where the text will be displayed. The Edit button will be added to the bottom of the field and clicking it will open a default text editor for editing the value.

```
f001 = formonly.f001 TYPE TEXT, PROGRAM = "txt";
```

Example

This example uses large variables and illustrates how they can be edited and displayed.

```
#####
#This example illustrates how TEXT variables are managed
#####

MAIN

DEFINE
    v_text      TEXT,      -- declaring TEXT variables
    v_byte      BYTE,
    answer      CHAR(3),   -- We will use this variable to control
    loc_status  CHAR(6),   -- the program workflow
    file1       VARCHAR(15),
    info,
    loc_mode,
    var_type    STRING,
    i           INT

OPTIONS
    HELP FILE "helpfile.erm"
```



```

# We open a window with the first form. Here we locate the text variables
# in named files or in the memory.
OPEN WINDOW text_win AT 2,2 WITH FORM "large_var_form" ATTRIBUTE (BORDER)

DISPLAY "!" TO loc_but
DISPLAY "!" TO value_but
DISPLAY "!" TO acc_but
DISPLAY "!" TO help_but1
DISPLAY "!" TO help_but3

INPUT BY NAME v_text, v_byte
# We specify actions that the program will take when the buttons on the form #
are pressed
ON ACTION ("locate")

LET var_type = fgl_winbutton("Locate Variable",
    "Which variable do you want to locate?", "TEXT",
    "TEXT|BYTE", "question", 1 )
IF var_type = "TEXT"
THEN
LET loc_mode = NULL

CALL fgl_winbutton("Locate Variable",
    "Where do you want to locate variable v_text?", "IN TEMP FILE",
    "IN TEMP FILE|IN FILE|IN MEMORY|FREE", "question", 1 ) RETURNING
loc_mode
CASE loc_mode
WHEN "IN TEMP FILE"
    IF i >1 THEN -- we check the counter indicating
        -- whether the variable is located
        -- in a named file
    FREE v_text
    LET i = 0 -- and reset it, because it was freed
        -- and is no longer located in a file
END IF
LOCATE v_text IN FILE -- the variable is located
        -- in a temporary file
DISPLAY "v_text is located IN TEMP FILE" TO file1 ATTRIBUTE (GREEN,
BOLD)
DISPLAY BY NAME v_text
WHEN "IN FILE"
    LET i=i+1 -- we st the counter to know
        -- how many times this option has been pressed
IF i>1 THEN
    FREE v_text -- we need to free the variable
        -- if it was previously located
        -- in a named file, before
        -- we can relocate it
END IF
INPUT BY NAME file1
ON KEY (F9)
    CALL showhelp(1)
END INPUT
LOCATE v_text in FILE file1
LET info = "v_text is located IN FILE ", file1

```



```

        DISPLAY info TO file1 ATTRIBUTE (GREEN, BOLD)
        DISPLAY BY NAME v_text
WHEN "IN MEMORY"
    IF i >1 THEN
        FREE v_text
        LET i = 0
    END IF
    LOCATE v_text IN MEMORY
    DISPLAY "v_text is located IN MEMORY" TO file1 ATTRIBUTE (GREEN, BOLD)
    DISPLAY BY NAME v_text
WHEN "FREE"
    FREE v_text
    DISPLAY "v_text was FREEd" TO file1 ATTRIBUTE (GREEN, BOLD)
WHEN NULL
    CALL fgl_winmessage("Choice Skipped",
        "The variable location was not changed.",
        "exclamation")

END CASE
END IF

IF var_type = "BYTE"
THEN
LET loc_mode = NULL

CALL fgl_winbutton("Locate Variable",
    "Where do you want to locate variable v_byte?", "IN TEMP FILE",
    "IN TEMP FILE|IN FILE|IN MEMORY|FREE", "question", 1 ) RETURNING
loc_mode
CASE loc_mode
WHEN "IN TEMP FILE"
    IF i >1 THEN -- we check the counter indicating
        -- whether the variable is located
        -- in a named file
        FREE v_byte
        LET i = 0 -- and reset it, because it was freed
        -- and is no longer located in a file
    END IF
    LOCATE v_byte in FILE -- the variable is located
        -- in a temporary file
    DISPLAY "v_byte is located IN TEMP FILE" TO file1 ATTRIBUTE (GREEN,
BOLD)
    DISPLAY BY NAME v_byte
WHEN "IN FILE"
    LET i=i+1 -- we st the counter to know
        -- how many times this option has been pressed
    IF i>1 THEN
        FREE v_byte -- we need to free the variable
            -- if it was previously located
            -- in a named file, before
            -- we can relocate it
    END IF
INPUT BY NAME file1
ON KEY (F9)
    CALL showhelp(1)

```



```
END INPUT
LOCATE v_byte IN FILE file1
LET info = "v_byte is located IN FILE", file1
DISPLAY info TO file1 ATTRIBUTE (GREEN, BOLD)
DISPLAY BY NAME v_byte
WHEN "IN MEMORY"
    IF i >1 THEN
        FREE v_byte
        LET i = 0
    END IF
    LOCATE v_byte IN MEMORY
    DISPLAY "v_byte is located IN MEMORY" TO file1 ATTRIBUTE (GREEN, BOLD)
    DISPLAY BY NAME v_byte
WHEN "FREE"
    FREE v_byte
    DISPLAY "v_byte was FREEd" TO file1 ATTRIBUTE (GREEN, BOLD)
WHEN NULL
    CALL fgl_winmessage("Choice Skipped",
                        "The variable location was not changed.",
                        "exclamation")

    END CASE
END IF
ON ACTION ("value")
    # We pass the TEXT value to a function by reference
    # and get it changed even though the function returns nothing
    CALL assign_value(v_text)

ON ACTION ("finish")
    LET answer = fgl_winquestion("QUIT?", "Do you really want to
quit?", "Yes", "Yes|No", "question", 1)
CASE
    WHEN answer MATCHES "Yes"
        EXIT INPUT
    WHEN answer MATCHES "No"
        CONTINUE INPUT
END CASE
# Pressing buttons F1, F2, and F3 will invoke the showhelp() function with
# corresponding help messages from the default help file.
# The numbers of these messages are passed to the function as its argument.
ON KEY (F1)
    CALL showhelp(1)
ON KEY (F3)
    CALL showhelp(2)

AFTER INPUT
    CONTINUE INPUT

END INPUT

CLOSE WINDOW text_win

END MAIN
```



```
#####
# This function assigns some value to a TEXT variable
# without using input from a form field.
# This illustrates how large variables are passed to functions.
#####
FUNCTION assign_value(p_text)
    DEFINE p_text TEXT,
        value_char CHAR(3000)

    # The user can assign a value to a character variable
    LET value_char = fgl_winprompt(8,10,"Enter value for variable v_text", "", 3000, 0)
    # Then this value is reassigned to the TEXT variable,
    # because we cannot input the value directly into
    # this variable from a field without the PROGRAM attribute
    LET p_text = value_char
    DISPLAY p_text TO v_text -- the value assigned to the TEXT variable
                           -- is displayed to the field, but is truncated
                           -- to fit the field
END FUNCTION
```

The Form File

This is the form file called "large_var_form.per". You can locate large variables, edit their values and display it them using it.

```
DATABASE foronly
SCREEN {
    [f10] [f12]
\gp-----q\g
| [f01] [f03] ] | \g
| [f02] ] | \g
\g \gBYTE variable: [f07] ] | \g | \g
| [f07] ] | \g
\g \gTEXT variable: [f08] ] | \g
| [f08] ] | \g
\gb-----d\g
}
}

ATTRIBUTES
f01 = foronly.loc_but, config="Locate {Locate:}", widget="button",
      LEFT, COLOR = BLUE BOLD;
```



```
f03 = formonly.file1;
f02 = formonly.value_but, widget="button",
      config="value {Enter value}", LEFT, COLOR=BLUE BOLD;
f08 = formonly.v_text TYPE TEXT, PROGRAM = "txt";
      -- this field will have the Edit button
f07 = formonly.v_byte TYPE BYTE, PROGRAM = "bmp";
      -- this field will have the Edit button
f05 = formonly.acc_but, widget = "button", config ="Finish {Finish}",
      COLOR = RED BOLD REVERSE, CENTER;
f10 = formonly.help_but1, widget="button", config = "F1 {Help(Locating)}",
      COLOR=YELLOW BOLD REVERSE;
f12 = formonly.help_but3, widget = "button", config = "F3 Help(Editing)",
      COLOR = BOLD YELLOW REVERSE, CENTER;
```

INSTRUCTIONS

DELIMITERS "[]"

The Help File

Here are the contents of the help file "helpfile.erm".

.1

Variable v_text_3 will be located IN MEMORY by default if no file name is provided for it and the LOCATE button is pressed.

Note that you must locate at least one variable in a file or in the memory in order to continue working with the program.

.2

To edit a value of the v_text variable, use the SHIFT-! key combination. The first time you do it, you will be offered to select the program to be used for editing, after that the selected program will be opened automatically when you relaunch the program and try editing again. Typically you would select the notepad program, if you work on Windows, which is located in C:\Windows directory. If it is the Paint you want to launch, it can be located in C:\Windows\System32

In the text editor that opens, enter a value, save the file and close it.

To edit the value of the BYTE variable, click the Edit button below the field. This button appears, if the PROGRAM attribute of the field has "bmp" value. It also allows to view the value.



Advanced Input Options

We have discussed the basic input methods in previous chapter. In this chapter, we'll go into details of the INPUT statement application and discuss some options and built-in functions that help the user in performing input.

The INPUT BY NAME Statement

4GL supports the BY NAME clause of the INPUT statement that binds the form fields to the variables implicitly, provided that the variable names and the field names match. In this respect, the INPUT BY NAME statement is close to the [DISPLAY BY NAME](#) statement that was described in Chapter 15.

The variables used for the input must match the names of the fields in the form file. They must also match in number and order. When the INPUT BY NAME statement is matching the field and the variable names, it ignores record or array name prefixes, if any. They can be added to the field names, but will be ignored.

The generalized syntax of the INPUT BY NAME statement is as follows:

```
INPUT BY NAME variable [, variable...]
```

Here "variable" is the name of a variable which has a matching field with the same name. You can use only the names of variables here and not the names of the fields. This means that though the names are the same, any record prefixes must be of program records and not screen records. The names of the variables and of the fields must not only match, but also be unique and unambiguous in order to prevent errors and resulting invalid values. If they do not meet these requirements, a compile-time error occurs.

For example, if a screen form has fields, named *f_name*, *l_name*, *dt*, and they belong to the screen record named *scr_rec*; and the program has variables named *f_name*, *l_name*, *dt*, that belong to the program array *pr_rec*, the following method can be used:

```
INPUT BY NAME pr_rec.f_name, pr_rec.l_name, pr_rec.dt
```

The line above is synonymous to the following line:

```
INPUT BY NAME f_name, l_name, dt
```

However, the following code will cause the input to fail, because the screen record prefix is used instead of the program record one:

```
INPUT BY NAME scr_rec.f_name, scr_rec.l_name, scr_rec.dt
```

You should not mix up the names of variables and fields even if they are the same.

You can also use the THROUGH keyword to indicate the consequent set of record members which are to receive values from the corresponding fields.



The INPUT OPTIONS

You can specify some settings for the INPUT statement by means of the OPTIONS statement. These settings influence the cursor behaviour and field order.

INPUT WRAP/INPUT NO WRAP

INPUT WRAP clause of the OPTIONS statement is used to allow the cursor wrap from the last to the first active field when the INPUT, INPUT ARRAY, or CONSTRUCT statements are being executed.

The default value of this option is INPUT NO WRAP. The INPUT NO WRAP clause of the OPTIONS statement indicates that the cursor won't wrap from the last form field to the first one, and the form will be deactivated as soon as the cursor becomes positioned to the last field and the user presses the RETURN or the ACCEPT key.

You can specify the INPUT WRAP option to change this behaviour:

```
OPTIONS INPUT WRAP
```

When this option is active, the cursor wraps from the last to the first field specified in these statements until the user presses the Accept key. If the user presses the RETURN key when the cursor is positioned to the last field of the screen form, the form won't be deactivated, and the cursor will move to the first form field again.

FIELD ORDER

You can use the OPTIONS statement to specify one of the following two options for the field order:

- FIELD ORDER CONSTRAINED means that when the user inputs values to the form fields, the UP arrow key will move the cursor to the previous field, and the DOWN arrow key will move the cursor to the next field.
- FIELD ORDER UNCONSTRAINED means that when the user inputs values to form fields, the UP arrow key will move the cursor to the field which is graphically placed above the current one, and the DOWN arrow key will move the cursor to the field that is placed below it.

You can change the field order by means of the *fgl_dialog_setfieldorder()* function which has the same effect as the OPTIONS FIELD ORDER statements. The function can take one argument, which is 1 for the constrained field order and 0 for the unconstrained one. The following two lines have the same effect:

```
OPTIONS FIELD ORDER CONSTRAINED  
CALL fgl_dialog_setfieldorder(1)
```

There are two sets of events that occur when the cursor changes its position. By default, when a cursor jumps from one field to another, the program activates some before and after logic for all the fields that are between these two. For example, if you have a form with three fields (f001, f002, f003), and you click the field f003 making the cursor jump there from the field f001, the program will perform the after logic of the



field f001, the before and after logic of the field f002, and the before logic of the field f003. You can see how it works in the following example:

```
MAIN

DEFINE v,v2,v3 CHAR(10)

OPEN WINDOW w1 AT 2,2 WITH FORM "myf"

ATTRIBUTES (Border)

INPUT v,v2,v3 FROM formonly.f001, formonly.f002, formonly.f003

BEFORE FIELD f001
    MESSAGE "Before Field F001"
    SLEEP 2

AFTER FIELD f001          --line a
    MESSAGE "After field F001"
    SLEEP 2

BEFORE FIELD f002
    MESSAGE "Before Field F002"
    SLEEP 2

AFTER FIELD f002
    MESSAGE "After field F002"
    SLEEP 2

BEFORE FIELD f003
    MESSAGE "Before Field F003"
    SLEEP 2          -- line b

AFTER FIELD f003
    MESSAGE "After field F003"
    SLEEP 2

END INPUT
END MAIN
```

When you run the program and click on the field f003, you'll have to read all the after field and before field messages that are between the lines *a* and *b*.

You can use the *fgl_dilog_fieldorder()* function to change this behaviour and to make the program activate only after logic of the current field and before logic of the destination field.

The function can have only one argument, which is TRUE or FALSE expressed by SMALLINT values 1 and 0, respectively. If you want the program to activate only the after logic of the current field and the before logic of the destination field, you have to pass "0" to the function. If you want the function to be applied to all the input, it should be called from the BEFORE OR AFTER block of the field to where the cursor is initially positioned or from the BEFORE INPUT control block.

```
BEFORE INPUT
    CALL fgl_dialog_fieldorder(0)
```



You can set the field order to the default by passing "1" to the function within one of the control blocks of the INPUT statement:

```
BEFORE INPUT
    CALL fgl_dialog_fieldorder(0)
    ....
    .....
BEFORE FIELD f003
    CALL fgl_dialog_fieldorder(1)
```

In this example, the program will execute only the after logic for the current field and the before logic for the destination field until the cursor is positioned to the field f003. When it happens, the program applies the default settings to the process of input.

Built-in Functions Used During Input

4GL supports a set of functions that can be used to estimate the results of the user input and to process the entered values. Here we will discuss some built-in functions that can prove useful when performing input into form fields.

The Functions Checking the Fields Data

There is a set of functions which deal with the form fields. These functions check the position of the cursor, the values entered to the fields and trace the changes that have been performed by the user during the input.

Field_touched()

The *field_touched()* function is used to check whether the user has entered or edited the value in the specified field or set of fields. The function can be used only within the INPUT statement, which we have already discussed, and within INPUT ARRAY and CONSTRUCT statements, which will be discussed in following chapters.

The general syntax of the *field_touched()* function invocation is as follows:

```
field_touched (field clause)
```

The *field clause* can include reference to one or several fields of the current screen form that belong to the same or different screen arrays or screen records. The field names can be preceded by array or record prefixes, e.g.: screen_record.field_name, screen_array.field name.

If the field isn't connected to any database and doesn't belong to any screen array or record, you can refer to the form field as formonly.field_name, or omit the prefix altogether.

The *field_touched()* function returns the TRUE value, if the user has changed the value displayed by the DISPLAY statement to the specified form field. The value is also considered to be changed, if the user has pressed one or several of the following keys while the cursor was positioned in the specified field:

- CONTROL-X - (delete a character);
- CONTROL-D - (clear the field);
- Any printable character, including SPACEBAR



If the user presses any of these keys, the FIELD_TOUCHD() function returns the TRUE value, irrespective of whether the value has actually changed.

If the user doesn't change the value displayed to the form field, the *field_touched()* function returns FALSE.

The result of the *field_touched()* function operation does not depend on the statements used in the BEFORE INPUT clause. If the statements in this clause display some values to form fields, it does not mean that these fields will be marked as touched.

The following example checks whether any values have been changed by the user. If not, the program asks the user whether they want to accept the values assigned during the program execution:

```
INPUT f_name,l_name,dat FROM formonly.f_name,formonly.l_name,formonly.dat

BEFORE INPUT
DISPLAY f_name, l_name, dat TO formonly.* -- displayed values are assigned
                                         -- by the program
SLEEP 3 -- the user is allowed to see the values

AFTER INPUT --checks the result of the input

# The following statement checks if any value displayed to the fields was
# changed

IF NOT field_touched(formonly.f_name, formonly.l_name, formonly.datum)
THEN

    PROMPT "Do you want to accept these values? (y/n)" FOR CHAR y_n

    IF y_n MATCHES "[Nn]" THEN

        CONTINUE INPUT

    END IF

END IF

END INPUT
```

Infield()

The *infield()* function is used when the choice of the following program actions depends on which field is the current one. The function checks whether its operand matches the name of screen field currently used for input.

The syntax of the function invocation is as follows:

```
infield (field_name)
```

The *infield()* function returns the value TRUE if the *field name* matches the name of the current field (do not confuse with the field tag!). Otherwise, the value returned by the *infield()* function is FALSE.



The *infield()* function is commonly used within the ON KEY clause.

The following example opens a new window with a message, if the user presses CONTROL-b when the cursor is positioned to the field *f_name*:

```
INPUT f_name, l_name FROM formonly.f_name, formonly.l_name

ON KEY (CONTROL-b)

IF infield(f_name) THEN
    OPEN WINDOW hint AT 2,3 SIZE 5,43 ATTRIBUTE (border)

DISPLAY "Enter your name, surname, and current date" AT 3,2

SLEEP 2
CLOSE WINDOW hint

END IF
END INPUT
```

Fgl_dialog_infield()

The *fgl_dialog_infield()* function is used to check whether the cursor is currently located within the field specified as the function argument. The function is very similar to the one described above. It returns the Boolean TRUE value if the cursor is in the specified field. If the cursor is not, the function returns FALSE.

The function is useful when the actions performed by the program depend on the current position of the cursor. In the following example, the program displays a message informing the user of the name of the current field (provided that the form contains three input fields), when they press the F2 button:

```
INPUT rec.* FROM formonly.*

ON KEY ("F2")

IF fgl_dialog_infield("f001") THEN MESSAGE "You're in field f001"
ELSE
    IF fgl_dialog_infield("f002") THEN MESSAGE "You're in field f002"
    ELSE MESSAGE "You're in field f003"

END IF

END IF

END INPUT
```



Retrieving Information From the Field Buffer

When the user types some value to a form field, this value is passed to the buffer and the program waits for the user to press the Accept or the Return key before it passes the entered value to the variable. There are some cases, when you need to refer to the field buffer, to check or to change the data stored in it. The following functions will help you to deal with these tasks.

Get_fldbuf()

The `get_fldbuf()` function is used to return the character values of the data entered to the specified fields of the current form. It can return the data entered into a field when no Accept key has yet been pressed or after it has been pressed. The function can appear only within the INPUT, INPUT ARRAY, and CONSTRUCT statements.

You can use the `get_fldbuf()` function to return values from one or from several fields. The syntax of the function invocation depends on the number of the returned values.

Retrieving the Value of a Single Field

If you use the `get_fldbuf()` function to return the value of one field, follow the syntax:

```
get_fldbuf(field_name)
```

The field name can be optionally preceded by a screen record/screen array prefix:

```
get_fldbuf(formonly.field_1)
```

If the field identifier is unambiguous, you can omit the record/array prefix in the field reference. In the following example, the program assigns the value of the field `f_name` to a variable which wasn't specified in the binding clause:

```
INPUT ph_addr, leg_addr, pos_code FROM adr_rec.*  
AFTER INPUT  
LET zipcode=get_fldbuf(pos_code)  
END INPUT
```

Retrieving Values From Several Fields

If you use the `get_fldbuf()` function to retrieve values from several fields, the syntax should be as follows:

```
CALL get_fldbuf(field_clause) RETURNING variable_list
```

The `field clause` can contain two or more references to fields that may belong to different screen records or screen arrays (they may be preceded by the record or array prefixes, e.g.):

```
CALL get_fldbuf(formonly.field_1, my_arr.field_2, my_rec.field_9)...
```



The variable list of the RETURNING clause may include variable or/and program record identifiers. The number of the fields in the field clause must correspond to the general number of variables listed in the RETURNING clause variable list, or to the number of record members, if a record is specified there.

The *get_fldbuf()* function in use may look as follows:

```
CALL get_fldbuf(my_arr.f001, my_arr.f003, formonly.f007)
      RETURNING a, my_rec.*
```

This statement will return a value to a program variable and to a program record, which consists of two members.

If the *get_fldbuf()* function assigns a character string value to a variable that is not of character data type, the program will try to convert the value to the necessary data type. The conversion can't be performed if the field contains convertible special characters, such as the currency character (\$).

In the following example this function is used to retrieve a value from a field while the input is not yet finished:

```
INPUT BY NAME my_rec.*
BEFORE INPUT
DISPLAY "Press F1 to verify whether the login is available" AT 2,2
ON KEY (F1)
LET my_rec.login=get_fldbuf(app_form.login_field)
IF my_rec.login = reg_log1 OR reg_log2 OR reg_log3 THEN
    DISPLAY "Such login is already in use" AT 2,2
    SLEEP 1
    DISPLAY "                                     " AT 2,2
    CONTINUE INPUT
ELSE
    DISPLAY "This login is available" AT 2,2
    NEXT FIELD app_form._pwd_field
END IF
AFTER INPUT
IF my_rec.login = reg_log1 OR reg_log2 OR reg_log3 THEN
    DISPLAY "You must enter the correct login" AT 2,2
    SLEEP 1
    DISPLAY "                                     " AT 2,2
    CONTINUE INPUT
END IF
END INPUT
```



In the example above the field contents is verified before the input is finished. After the Accept key is pressed, we do not need to use this function to get the field contents, as its value is already assigned to the corresponding variable used for input.

Fgl_dialog_getbuffer()

As it has been said above, when the user enters a value to a form field, this value is stored to the buffer and only then is passed to the variable linked to this field. The value is normally passed to the variable when the user presses the Accept key or when he moves the cursor off the field. If, for any reason, you need to use the content of the field and cannot or it is not necessary to write the value to the corresponding variable, you can use the *fgl_dialog_getbuffer()* function.

In the following example, the user can check if they have entered the correct value during the input before they press the Accept key and the value is passed to the variable:

```
INPUT val FROM f001

    BEFORE INPUT

        MESSAGE "Input a positive number"

        ON KEY (F2)

            LET val2 = fgl_dialog_getbuffer()

            IF val2>0 THEN

                MESSAGE "The value is correct   "

            ELSE

                MESSAGE "The value is incorrect"

            END IF

        END INPUT
```

When the user enters a value to the field f001, they can press the F2 key to know whether the value can be accepted by the program. In this case, the program evaluates the data that is currently typed to the field named f001.

Fgl_dialog_setbuffer()

The *fgl_dialog_setbuffer()* function has no return value. The argument of the function is a value of a variable. This value will be set to the current field.

You can invoke the function as *fgl_dialog_setbuffer(variable)* or as *dialog.setbuffer(variable)*.

When using the function, you have to remember the following rules and descriptions:

- The program does not execute the *fgl_dialog_setbuffer()* function if it is invoked from the BEFORE FIELD clause



- If the function is called from the INPUT statement, it modifies the input buffer of the current field. The input variable is still assigned when the cursor leaves the field, not when the *fgl_dialog_setbufer()* function is executed.
- If the function is executed, the program marks the current field as "touched", therefore, the *field_touched()* function returns TRUE.

In the following example, the user is to input a value for the variable *val2*. If they press the F2 key, the input field will be filled with the value from the input buffer which is taken from the variable *val*.

```
DEFINE val, val2 INT
LET val = 123
OPEN WINDOW w1 AT 3,3 WITH FORM "myf"
ATTRIBUTES (border)
INPUT val2 FROM f001
    BEFORE INPUT
        MESSAGE "Press F2 for a default value"
    ON KEY (F2)
        CALL fgl_dialog_setbuffer(val)
END INPUT
```

Fgl_buffertouched()

The *fgl_buffertouched()* function returns an integer value 1 or 0 (which corresponds to the Boolean TRUE or FALSE) which indicates whether the user or the program has modified the last field.

You should use the function in user interaction statements such as INPUT statements, DISPLAY ARRAY, or PROMPT. If this function is used in other contexts, it can result in compile or runtime errors. In the following example, the program uses the AFTER INPUT block to check whether the user has entered any values. If the user enters not all the values, a message will appear and after that the input will be continued. If all the values are entered, the program accepts them and passes to the corresponding variables:

```
MAIN
DEFINE val, val2, val3 CHAR(10),
chk INT
OPEN WINDOW w1 AT 3,3 WITH FORM "myf"
ATTRIBUTES (border)
INPUT val, val2, val3 FROM f001, f002, f003
AFTER INPUT
```



```
LET chk = fgl_buffertouched()

IF chk = 0 THEN

    CALL fgl_winmessage("!", "One or more fields are without
values", "exclamation")

ELSE

    EXIT INPUT

END IF

CONTINUE INPUT

END INPUT

CLOSE WINDOW w1

CALL fgl_winmessage("!", "The input has been performed
successfully ", "information")

END MAIN
```

Functions that Return Key Values

There are several functions in 4GL which return some values depending on the key the user pressed. There are *fgl_keyval()*, *fgl_getkey()*, and *fgl_lastkey()*. They may prove useful during the input, as they help to monitor which keys the user presses. There are also functions that pass a keypress to the program without a real key being pressed. Thus a program can be instructed to behave in different ways depending on the keys the user presses.

Fgl_getkey()

The *fgl_getkey()* function returns the integer code of a key pressed by the user. This function does not accept any arguments, the syntax of its invocation is:

```
fgl_getkey()
```

The *fgl_getkey()* function is case-sensitive. The value returned by the user is a raw value not of the character, but of the physical key pressed by the user. The function can also return integer codes of single-byte non-ASCII characters from the locale code set.

You can invoke the *fgl_getkey()* function from any place of the program. This simple example will display the code of the key pressed by the user:

```
MAIN

DEFINE key_code INTEGER

DISPLAY "Press any key" at 2,2

LET key_code = fgl_getkey() -- here the program will wait for a keystroke
```



```
DISPLAY "The code of the key pressed is: ", key_code AT 2,2  
END MAIN
```

When 4GL encounters this function, it waits for a keystroke to be performed. Thus the application does not have to be in the input mode for the user to be able to press a key.

Fgl_lastkey()

The *fgl_lastkey()* function is used to return an integer code of the last key pressed by the user during the value input to the form field. The *last* key means the key that the user pressed the last before the *fgl_lastkey()* function was invoked. This function does not require arguments as well. The syntax of the function invocation is:

```
fgl_lastkey()
```

The function returns integer values for all the ASCII characters. This simple example will illustrate the *fgl_lastkey()* function operation (you can use your own field and variable names instead of the given ones in order to check the function operation in your form file):

```
INPUT f_name, l_name FROM formonly.f_name, formonly.l_name  
AFTER INPUT  
  
DISPLAY "The code number of the key you pressed the last is ",  
fgl_lastkey() at 25,2  
  
END INPUT
```

If the *fgl_lastkey()* function is invoked in the MENU statement, the result of the function execution will be undefined.

Fgl_keyval()

Function *fgl_keyval()* returns the integer code of a logical or physical key. It requires an attribute which must be a character expression enclosed in quotation marks:

```
FGL_KEYVAL( "character expression")
```

The character expression must specify:

- A single letter or digit - the case is considered
- Non-alphabetic symbols (i.e. !, @, ^)
- A keyword specifying a logical key (ACCEPT, DELETE, DOWN, ESC/ESCAPE, F1-F36, HELP, INSERT, INTERRUPT, LEFT, NEXT, PREVIOUS, RETURN, RIGHT, TAB, UP, CONTROL-*character*) - the case is ignored

As a rule, the *fgl_keyval()* function is used to examine values returned by the *fgl_getkey()* and *fgl_lastkey()* functions. The *fgl_keyval()* function is typically used in a conditional statement to ensure that the key pressed by the user and then returned by the *fgl_getkey()* or *fgl_lastkey()* function corresponds to the key required:



```
...
INPUT BY NAME clients.*

...
AFTER FIELD company
IF fgl_lastkey() = FGL_KEYVAL("d") THEN
...
END IF
END INPUT
```

You can also find out whether a user performed some actions rather than entered a value (i.e. whether a user pressed the Accept key). In such case you should use a name of a logical key as the function argument (,e.g. *fgl_keyval(ESC)*).

Fgl_keyname()

The *fgl_keyname()* function is the reverse function of the *fgl_keyval()* function. The function returns a VARCHAR value which indicates the keyname corresponding to the key value specified as the function argument.

In the following example, the *fgl_keyname()* function returns "accept":

```
MAIN

DEFINE i INTEGER
DEFINE c CHAR(20)

LET i = fgl_keyval("Escape")
LET c = fgl_keyname(i)
DISPLAY "The key is", c
SLEEP 2
END MAIN
```

Fgl_key_queue()

The *fgl_key_queue()* function is used to send an argument key value to the key queue. If the key queue exists, the program automatically returns the key at its top when the *fgl_getkey()* function is called. The argument of the *fgl_key_queue()* function is an integer number, representing the key value.

In the following example, the *fgl_key_queue()* function passes a key to the key queue, and this key is returned by the *fgl_getkey()* function:

```
MAIN
```



```
DEFINE my_key INT
CALL fgl_key_queue(50) -- Push the key 'a'
CALL fgl_getkey()      -- into the queue
LET my_key = fgl_lastkey() -- retrieves the value of the last key
buffer
DISPLAY "You pressed the key: ", my_key at 5,5
SLEEP 3
END MAIN
```

Function Button Field

The function button field is a standard field with a button displayed to the right. The button is used to trigger a key or an event and has a bitmap image specified as a *config* value in the ATTRIBUTES section of the form file.

An example of the function button field application would be a field with an internet address displayed to it. The button can be used to open an internet-window with the corresponding URL.

The attribute syntax of the function button field is as follows:

```
field_tag = formonly.field_name
config   = "button_image Key/Action"
widget   = "field bmp";
```

In the following example, we have a form file containing a browser field, a button field tagged *Close*, and a function button field.

The function button field is used for input of new URL addresses that are sent to the browser field. The button to the right from the input field is used to return to the homepage, which is set as *www.querix.com*.

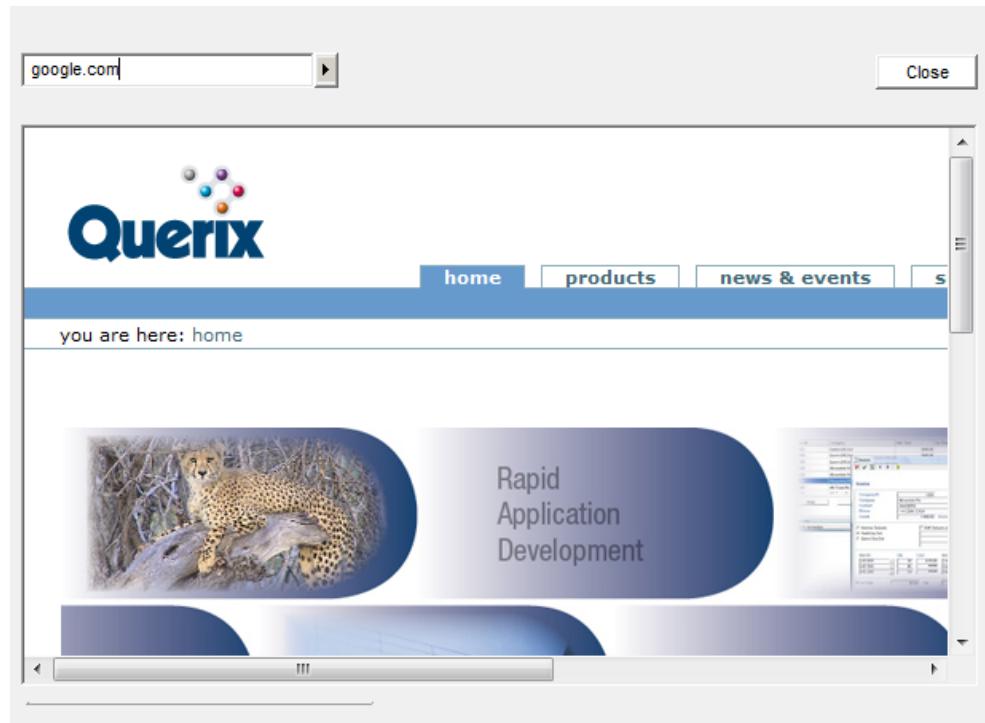
Form file attributes:

```
ATTRIBUTES
my_but = formonly.mbutton,
config = "end Close",
widget = "button";
funf = formonly.func_f,
config = "browsr",
widget = "field bmp";
brwin = formonly.mybrs,
config = "www.querix.com",
```



```
widget = "browser" ;  
  
INSTRUCTIONS  
DELIMITERS "||"  
  
Executable file:  
  
MAIN  
  
DEFINE newurl VARCHAR(20)  
  
OPEN WINDOW w1 AT 2,2 WITH FORM "myform"  
DISPLAY "!" TO mbbutton  
  
LABEL inp:  
INPUT newurl FROM func_f  
ON ACTION ("browsr")  
DISPLAY "www.querix.com" TO mybrs  
ON ACTION ("Close")  
EXIT PROGRAM  
END INPUT  
DISPLAY newurl TO mybrs  
GOTO inp  
  
END MAIN
```

When you run the application, the program will connect to the www.querix.com website as it is the default page for the browser widget. The user will be able to enter a new URL to the function button field. When they click the button to the right from the button field, the browser will reconnect to the Querix.com.



Example

The example below illustrates the INPUT BY NAME variant of the INPUT statement as well as gives a wider overview of the built-in functions typically used during the input which includes the functions used to work with the field buffer.

```
#####
# This example illustrates the INPUT BY NAME statement and built-in functions
#####
DEFINE
    n_key, constr      INTEGER,
    answer            CHAR(3),
    acc_date          VARCHAR(15),
    r_all_data RECORD
        id              INTEGER,
        first_name     CHAR(30),
        last_name      DATE,
        age             INTEGER,
        phone           CHAR(13),
        units_pd        DECIMAL(8,2),
        price_pu        MONEY(16,2),
        id_card         CHAR(10),
        date_issue     DATE,
        date_expire    VARCHAR(80),
        address         CHAR(20),
        city            CHAR(6),
        zip_code        CHAR(6),
```



```

        country_name   CHAR( 20 ),
        add_data       CHAR(512),
        start_date     VARCHAR(15),
        start_time     DATETIME HOUR TO SECOND,
        end_date       DATE,
        end_time       DATETIME HOUR TO SECOND,
        accept_date    VARCHAR(15)
    END RECORD

```

MAIN

```

OPTIONS
    ACCEPT KEY (F3),           -- This option sets the F3 as the Accept key
                                -- instead of the Escape key

    FIELD ORDER CONSTRAINED,   -- This option is applied by default even if we
                                -- do not specify it explicitly.
                                -- You can move the cursor from field to field
                                -- only in the order in which they are listed in
                                -- the ATTRIBUTES section

    HELP FILE "my_help.erm"

IF fgl_fglgui()= 0 THEN
    CALL fgl_winmessage("Wrong mode",
                        "Run this application in the GUI mode.", "info")
    EXIT PROGRAM
END IF

# We initialize some members of record r_all_data
LET r_all_data.id                  = 1
LET r_all_data.first_name          = "Thomas"
LET r_all_data.last_name           = "Jones"
LET r_all_data.date_birth          = "03/08/1965"
LET r_all_data.age                 = YEAR(TODAY) -
                                         YEAR(r_all_data.date_birth)
LET r_all_data.phone               = "023-456-78-99"
LET r_all_data.units_pd            = 5.5
LET r_all_data.price_pu            = 9.00
LET r_all_data.id_card             = "2397213D"
LET r_all_data.date_issue          = "02/01/2000"
LET r_all_data.date_expire         = "02/01/2015"
LET r_all_data.address             = "102, Everton Rd."
LET r_all_data.city                = "Southampton"
LET r_all_data.zip_code            = "123456"
LET r_all_data.country_name        = "United Kingdom"
LET r_all_data.add_data            =

"Here the additional comments are displayed to a multisegment field. ",
"This field can comprise several lines of text either displayed to it",
"or entered by a user. In a form file a multi-segment field looks like",
"several fields with the same tag placed one under another. ",
"The WORDWRAP attribute is required to create a multi segment field."
LET r_all_data.start_date          = TODAY

```



```

LET r_all_data.start_time      = CURRENT HOUR TO SECOND
LET r_all_data.end_date        = "12/31/9999"
LET r_all_data.end_time        = "23:59:59"
LET r_all_data.accept_date     = TODAY

MENU "Main"

COMMAND "fgl_dialog_fieldorder(1)"
        "The before/after field logic of all the fields in between is executed"

CALL fgl_winmessage("Field Logic",
        "It may take some time for the program to execute" ||
        "\nall the before and after field logic, so the execution" ||
        "\nmay be somewhat slow. The execution of the logic" ||
        "\ncan be viewed by observing the MESSAGES appearing" ||
        "\non the second line of the window when cursor moves.", "info")
CALL screen_1(1) -- this value, passed to the fgl_dialog_fieldorder
                  -- function makes the program
                  -- execute all the AFTER and BEFORE FIELD blocks
                  -- of all the fields between the current one
                  -- and the selected one

COMMAND "fgl_dialog_fieldorder(0)"
        "The before/after field logic of all the fields in between is not
executed"
        CALL screen_1(0) -- this value, passed to the fgl_dialog_fieldorder
                          -- function makes the program
                          -- execute only the AFTER FIELD block of the current
                          -- field and the BEFORE FIELD block of the
                          -- selected one

COMMAND "NO WRAP"   "NO WRAP and FIELD ORDER demo"
        CALL screen_2()
COMMAND "Buffer functions"
        "Applies functions used with field buffers"
        CALL screen_3()
COMMAND "Exit"      "Exit the demo program"
        EXIT PROGRAM

END MENU

END MAIN

#####
# The following function displays a form and activates the INPUT BY NAME
# statement with different control blocks
#####

FUNCTION screen_1(chk)
DEFINE
    chk    INTEGER
# First we display some auxiliary information to the form
    OPEN WINDOW w_app_form_3 AT 1,1 WITH FORM "app_form_3"
        ATTRIBUTE (FORM LINE 2)

```



OPTIONS INPUT WRAP

```
DISPLAY "!" TO hlp
DISPLAY "!" TO sav
display "!" to auto_input
DISPLAY "!" TO quit

# There is no list of form fields as their names coincide with the names of the
# record members
LET constr = 1
INPUT BY NAME r_all_data.* WITHOUT DEFAULTS

BEFORE INPUT
CALL fgl_dialog_fieldorder(chk) -- we apply the selected before/after field
logic

# When the user leaves the last field the cursor is returned to the first field
# and the INPUT remains active due to the WRAP option in effect

ON ACTION ("a_inpt")

# This ON ACTION clause will automatically fill in
# the country field usif the gfl_key_queue() function
CALL auto_input()
CONTINUE INPUT

ON ACTION ("hlp")
CALL showhelp(1) -- though the INPUT statement has no HELP clause, the
-- program will display the first help
message,
-- if F1 is pressed due to this function.
-- The same effect would be achieved, if the
INPUT
-- statement had HELP 1 clause

ON KEY (F3) -- as it is the Accept key, the data are saved
CALL fgl_winmessage("Save", "The data were saved", "INFO")

ON KEY (F10)
EXIT INPUT

# The following set of BEFORE and AFTER FIELD blocks specifies the actions
# to be taken before and after input to the first_name, last_name,
# and date_birth fields.
# These blocks are executed in different ways according to the selected
# option of the main menu
# The function clears the screen after the input and returns the program
# control to the main menu

BEFORE FIELD first_name
MESSAGE "Input the first name" ATTRIBUTE (RED, REVERSE)
SLEEP 1
AFTER FIELD first_name
MESSAGE "First name filled in" ATTRIBUTE (RED, REVERSE)
```



```

SLEEP 1
BEFORE FIELD last_name
MESSAGE "Input the last name" ATTRIBUTE (RED, REVERSE)
SLEEP 1
AFTER FIELD last_name
MESSAGE "Last name filled in" ATTRIBUTE (RED, REVERSE)
SLEEP 1
    BEFORE FIELD date_birth
MESSAGE "Input the date of birth" ATTRIBUTE (RED, REVERSE)
SLEEP 1

AFTER FIELD date_birth
MESSAGE "Birth day filled in" ATTRIBUTE (RED, REVERSE)
SLEEP 1
IF r_all_data.date_birth IS NOT NULL AND
    r_all_data.date_birth > "12/31/1899"
THEN LET r_all_data.age = YEAR(TODAY) - YEAR(r_all_data.date_birth)
ELSE LET r_all_data.age = NULL
END IF
DISPLAY BY NAME r_all_data.age

AFTER INPUT
# This clause is executed when the Accept key(F3) is pressed,
# or if you press F10 to leave the form,
# while the ON KEY (F10) clause redirects the program workflow here
LABEL lab_exit:

# The IF statement checks whether the value in at least one of the form
# fields has been modified
IF FIELD_TOUCHED(first_name) OR
    FIELD_TOUCHED(last_name) OR
    FIELD_TOUCHED(date_birth) OR
    FIELD_TOUCHED(phone) OR
    FIELD_TOUCHED(units_pd) OR
    FIELD_TOUCHED(price_pu) OR
    FIELD_TOUCHED(id_card) OR
    FIELD_TOUCHED(date_issue) OR
    FIELD_TOUCHED(date_expire) OR
    FIELD_TOUCHED(address) OR
    FIELD_TOUCHED(city) OR
    FIELD_TOUCHED(zip_code) OR
    FIELD_TOUCHED(country_name) OR
    FIELD_TOUCHED(add_data) OR
    FIELD_TOUCHED(start_date) OR
    FIELD_TOUCHED(start_time) OR
    FIELD_TOUCHED(end_date) OR
    FIELD_TOUCHED(end_time) OR
    FIELD_TOUCHED(accept_date)
THEN          -- if the data of at least one field has been changed,
              -- the nested IF statement is executed

    IF FGL_LASTKEY() = FGL_KEYVAL("F10")
    THEN      -- if the last key pressed before the AFTER INPUT clause
              -- began executing was F10 the THEN clause is executed
        LET answer = fgl_message_box("The data were changed",

```



```
"Do you want to quit? (YES to terminate the input)", 4)
  IF answer = "1" THEN
    EXIT INPUT
  ELSE CONTINUE INPUT
  END IF

END IF

# The second nested IF statement of the THEN clause of the first
# level IF statement
IF FGL_LASTKEY() = FGL_KEYVAL("F3")

THEN      -- this clause is executed in the Accept key (F3) was
          -- the last key pressed before the AFTER INPUT clause
          -- started executing in this case some values are
          -- verified and saved in record r_all_data

# If the values have not been modified, the cursor is returned
# to the corresponding field and the input continues

IF GET_FLDBUF(first_name) IS NULL
  -- this function checks the buffer of the field
THEN ERROR " Enter a value in the field "
  NEXT FIELD first_name
END IF
IF GET_FLDBUF(last_name) IS NULL
THEN ERROR " Enter a value in the field "
  NEXT FIELD last_name
END IF
IF GET_FLDBUF(date_birth) IS NULL OR
  r_all_data.date_birth <= "12/31/1899"
THEN ERROR "The date must be later than 12/31/1899"
  NEXT FIELD date_birth
END IF
IF GET_FLDBUF(date_issue) IS NULL OR
  r_all_data.date_issue <= "12/31/1899"
THEN ERROR "The date must be later than 12/31/1899"
  NEXT FIELD date_issue
END IF
IF GET_FLDBUF(date_expire) IS NULL OR
  r_all_data.date_expire <= "12/31/1899"
THEN ERROR "The date must be later than 12/31/1899"
  NEXT FIELD date_expire
END IF

IF r_all_data.date_issue >= r_all_data.date_expire THEN
ERROR "The expire date must not be earlier than the date of
issue"
  NEXT FIELD date_issue
END IF

CALL fgl_winmessage ("Attention", "The data was saved", "exclamation")

CONTINUE INPUT -- after the data are saved the input remains
```



```

-- active and is not terminated
END IF

ELSE      -- This is the ELSE clause of the first level IF statement.
-- It is executed if none of the fields has been modified and
-- either ACCEPT or F10 is pressed
IF FGL_LASTKEY() = FGL_KEYVAL("F10")--IF nested in the ELSE clause

    THEN CALL fgl_winmessage("Attention",
    "No data was modified\nThe input will be terminated",
    "exclamation")
    EXIT INPUT      -- we exit from INPUT and the second input
                    -- screen is displayed
END IF
IF FGL_LASTKEY() = FGL_KEYVAL("F3")
    THEN CALL fgl_winmessage("Attention,
    You pressed Accept, but no data is modified\nPress OK to
continue",
    "exclamation")
    CONTINUE INPUT -- The INPUT is not terminated and the user
                    -- is allowed to modify fields
END IF
END IF

END INPUT

CLOSE WINDOW w_app_form_3
CLEAR SCREEN

END FUNCTION

#####
# The function demonstrates the NO WRAP and FIELD ORDER UNCONSTRAINED
# options
#####
FUNCTION screen_2()
DEFINE      chk INTEGER

OPTIONS
    INPUT NO WRAP,      -- though this option is the default one, we
                        -- need to specify it explicitly
                        -- to override the OPTIONS INPUT WRAP option
                        -- used above this option terminates the
                        -- INPUT when the cursor leaves the last form
                        -- field or is the Accept key is pressed

    FIELD ORDER UNCONSTRAINED      -- this function overrides the FIELD ORDER
                                    -- CONSTRAINED option specified earlier
                                    -- the cursor can be moved up and down using
                                    -- UP and DOWN arrow keys whereas if the
                                    -- order is constrained they move the cursor
                                    -- to the next or previous field only

OPEN WINDOW w_app_form_3 AT 1,1 WITH FORM "app_form_3"
ATTRIBUTE (FORM LINE 2)

```



```
DISPLAY "NO WRAP and FIELD ORDER options. Use arrow keys to move cursor:"  
TO header ATTRIBUTE(GREEN, BOLD)  
  
DISPLAY "!" TO hlp  
DISPLAY "!" TO sav  
DISPLAY "!" TO auto_input  
DISPLAY "!" TO qit  
  
INPUT BY NAME r_all_data.* WITHOUT DEFAULTS  
  
ON ACTION (hlp)  
  IF fgl_dialog_infield("first_name") THEN -- the first_name field  
    -- has its own help message  
    CALL showhelp(2)  
  ELSE CALL showhelp(1)  
  END IF  
  
ON KEY (F3)  
  LET chk = fgl_buffertouched()  
  IF chk = false THEN  
    LET answer = fgl_message_box("Not all fields are changed",  
    "Do you want to save the default values and exit?", 4)  
    IF answer = "1" THEN  
      CALL fgl_winmessage("Attention", "Default data were saved",  
      "Info")  
      EXIT INPUT  
    ELSE CONTINUE INPUT  
    END IF  
  ELSE  
    CALL fgl_winmessage("Attention", "The data were saved", "Info")  
    EXIT INPUT  
  END IF  
  
ON ACTION ("a_inpt")  
  
  # This ON ACTION clause will automatically fill in  
  # the country field usif the gfl_key_queue() function  
  CALL auto_input()  
  CONTINUE INPUT  
  
ON KEY (F10)  
  CALL fgl_winmessage ("Quit", "The input is over", "Info")  
  EXIT INPUT  
  
END INPUT  
  
CLOSE WINDOW w_app_form_3  
CLEAR SCREEN  
  
OPTIONS -- we reset the options after the function execution  
  INPUT WRAP,  
  FIELD ORDER CONSTRAINED  
END FUNCTION
```



```
#####
# This function illustrates the operation of buffer functions
#####
FUNCTION screen_3()

DEFINE
    fill_fname, fill_position, buf_fname, buf_position,
    buf_city, buf_positioncheck VARCHAR (20),
    fill_date, buf_date DATE,
    yno INTEGER

OPTIONS INPUT NO WRAP

OPEN WINDOW fill_data AT 2,2 WITH FORM "buffer_form"
ATTRIBUTES (BORDER)

DISPLAY "!" TO qit
DISPLAY "!" TO ent
DISPLAY "!" TO check_v

# These are the default values that can be displayed to the form fields
LET buf_fname = "Jack Jackson"
LET buf_position = "Human Relations"
LET buf_date = TODAY

INPUT fill_fname, fill_position, fill_date
    FROM first_name, position, f_date

# The ON KEY block is used to check whether the entered position name is
# available for the input
ON KEY (F5)

    IF infield (position) THEN
        LET buf_positioncheck = fgl_dialog_getbuffer()
        # We use the upshift() function to make sure the entered
        # value will be recognized regardless of the case
        LET buf_positioncheck = upshift(buf_positioncheck)
        IF buf_positioncheck = "HUMAN RELATIONS" OR
            buf_positioncheck = "MANAGER" OR
            buf_positioncheck = "C PROGRAMMER" OR
            buf_positioncheck = "QA" OR
            buf_positioncheck = "TEAM LEAD"
        THEN
            CALL fgl_winmessage("Correct",
                "The entered position is valid",
                "info")
            CONTINUE INPUT
        ELSE
            CALL fgl_winmessage("Enter the city name",
                "The entered position is not in the list" ||
                "\nThe possible positions are:" ||
                "\nHuman relations, Manager, C Programmer, QA, Team Lead",
                "error")
            CONTINUE INPUT
        END IF
    END IF
```



```

        ELSE
            NEXT FIELD position
        END IF

# The ON ACTION ("bufval") block displays different default values depending
# on the field where the cursor is currently located.
ON ACTION ("bufval")
    IF fgl_dialog_infield("first_name")
        THEN CALL fgl_dialog_setbuffer(buf_fname)
    ELSE IF fgl_dialog_infield("position")
        THEN CALL fgl_dialog_setbuffer(buf_position)
    ELSE CALL fgl_dialog_setbuffer(buf_date)
    END IF
END IF

# The first ON ACTION block specifies the actions that the program should
# perform when the user presses the "Back" button
ON ACTION (qit)
    LET yno = fgl_message_box("EXIT",
        "Do you want to return to the main menu?", 4)
    IF yno = 1 THEN
        EXIT INPUT
    ELSE CONTINUE INPUT
    END IF
# The AFTER INPUT control block displays the fill_fname and fill_lname values
# to the thenx field
AFTER INPUT

    DISPLAY "Thank you, "||fill_fname TO thanx
    ATTRIBUTES (RED)
CONTINUE INPUT
END INPUT

CLOSE WINDOW fill_data

END FUNCTION

#####
# This function allows the user to perform automatic input
# into some of the fields
#####
FUNCTION auto_input()

CASE
    WHEN INFIELD("country_name")
        CALL fgl_key_queue(85)
        CALL fgl_key_queue(83)
        CALL fgl_key_queue(65) -- results is "USA"
    WHEN INFIELD("first_name")
        CALL fgl_key_queue(74)
        CALL fgl_key_queue(111)
        CALL fgl_key_queue(104)
        CALL fgl_key_queue(110) -- results in "John"
    WHEN INFIELD("last_name")

```



```

        CALL fgl_key_queue(83)
        CALL fgl_key_queue(109)
        CALL fgl_key_queue(105)
        CALL fgl_key_queue(116)
        CALL fgl_key_queue(104) -- results in "Smith"
OTHERWISE
    CALL fgl_winmessage("Wrong Field",
        "The cursor must be located at one of these fields:" ||
        "\nCountry, First Name, Last Name", "Info")
END CASE

END FUNCTION

```

Form Files

The following form file is referenced by the functions sscreen_1() and screen_2 and is stored in the form file named 'app_form_3.per'

```

DATABASE formonly
SCREEN SIZE 55 BY 80
{
\g
\g [t01
\gp-----[t02 ]-----q\g
\g|\g      ID:\g[f00      ]
\g|\g  First Name:\g[f01      ]
\g|\g  Last Name:\g[f02      ]
\g|\g  Birth Date:\g[f05      ]\g Age:\g[f06 ]
|\g
\g|\g      Phone:\g[f07      ]
\g|\g Units per day:\g[f08      ]\g      Price per unit:\g[f09      ]
|\g
\g|-----[t03      ]-----| \g
\g|\g      ID Number:\g[f10      ]\g Issued on:\g[f11      ]\g Expires on:\g[f12      ]
|\g
\g|-----[t04      ]-----| \g
\g|\gAddress:\g[f13      ]\g Country:\g[f17
\g|\g City:\g[f14      ]\g Zip:\g[f15      ]
|\g
\g|-----[t05      ]-----| \g
\g|[f18      ]\g
\g|[f18      ]\g
\g|-----[t06      ]-----| \g
\g|\gRegistered:\g[f19      ][f20      ]\g Dismissed:\g[f21      ][f22      ]
|\g
\g|\g Accepted:\g[f23      ]
\g|  [t07      ]  [t08      ]  [t09      ]  [t10      ]  |\g
\gb-----d\g
}

```

ATTRIBUTES

```

t07 = formonly.hlp, config = "hlp {HELP}", widget = "button", COLOR = YELLOW;
t08 = formonly.sav, config = "F3 {SAVE}", widget = "button", COLOR = GREEN;

```



```
t09 = formonly.auto_input, config = "a_inpt {Auto Input}", widget = "button",
COLOR = GREEN;
t10 = formonly.qit, config = "F10 {BACK}", widget = "button", COLOR = RED;

f00 = formonly.id TYPE INTEGER,COLOR = YELLOW REVERSE,NOENTRY;
f01 = formonly.first_name TYPE CHAR,COMMENTS=" Enter the first name";
f02 = formonly.last_name,COMMENTS=" Enter the last name";
f05 = formonly.date_birth TYPE DATE, FORMAT="mm.dd.yyyy", PICTURE="##.##.####",
CENTURY = "R", COMMENTS=" Enter the date birth";
f06 = formonly.age TYPE INTEGER,NOENTRY;
f07 = formonly.phone,PICTURE = "###-###-##-##",
COMMENTS=" Enter the phone number";
f08 = formonly.units_pd TYPE DECIMAL,FORMAT="-,---,--&.&&",
COMMENTS=" Enter the units per day";
f09 = formonly.price_pu TYPE DECIMAL,INVISIBLE,
COMMENTS=" Enter the price per unit";
f10 = formonly.id_card TYPE CHAR,COMMENTS=" Enter the id card number";
f11 = formonly.date_issue, WIDGET = "calendar", AUTONEXT,
COMMENTS=" Enter the date the ID was issued";
f12 = formonly.date_expire, WIDGET = "calendar", AUTONEXT,
COMMENTS=" Enter the date when ID expires";
f13 = formonly.address TYPE VARCHAR,
COMMENTS=" Enter the street address, apartment, housing, etc.";
f14 = formonly.city TYPE CHAR,DEFAULT = "Southampton",
COMMENTS=" Enter the city name";
f15 = formonly.zip_code TYPE CHAR,REQUIRED,
COMMENTS=" Enter the zip code for this address";
f17 = formonly.country_name,
COMMENTS=" Enter the country for this address.";
f18 = formonly.add_data TYPE CHAR, WORDWRAP,
COMMENTS=" Enter the additional data";
f19 = formonly.start_date;
f20 = formonly.start_time TYPE DATETIME HOUR TO SECOND, DEFAULT = CURRENT,
NOENTRY;
f21 = formonly.end_date, WIDGET = "calendar", REQUIRED,
COMMENTS=" Enter the date of dismissing";
f22 = formonly.end_time TYPE DATETIME HOUR TO SECOND, REQUIRED,
COMMENTS=" Enter the time of dismissing";
f23 = formonly.accept_date, WIDGET = "calendar", REQUIRED,
COMMENTS=" Enter the date of accepting";

t01 = formonly.headr,
config = "The INPUT BY NAME statement is used with all the form fields",
COLOR=GREEN BOLD,
widget = "label", center;
t02 = formonly.apf,
config = " Application form ", REVERSE,COLOR=BLUE,
widget = "label", center;
t03 = formonly.idcard,
config = " ID card ", COLOR=BLUE,
widget = "label", center;
t04 = formonly.addr,
config = " Address ", COLOR=BLUE,
widget = "label", center;
```



```
t05 = formonly.adit,
    config = "Additional Data", COLOR=BLUE,
    widget = "label", center;
t06 = formonly.reg,
    config = "Registration", COLOR=BLUE,
    widget = "label", center;
```

INSTRUCTIONS

```
DELIMITERS "[ ]"
```

The following form is used by the function screen_3() and is stored in the form file named 'master_form'

```
DATABASE formonly
```

```
SCREEN
```

```
{
\gp-----
\g| [f001
\g|
\g| \g Name: \g      [f002      ]
\g|
\g| \g Position: \g   [f003      ] [f006      ]
\g|
\g| \g Date: \g       [f004      ]
\g|
\g| [f005
\g|
\g|
\g|
\g|           [f008      ] [f007      ]
\gb-----d\g
}
```

```
ATTRIBUTES
```

```
f002 = formonly.first_name,
    config = "icon.jpg bufval",
    widget = "field_bmp",
    COMMENTS = "Press the arrow to enter a value from the buffer";
f003 = formonly.position,
    config = "icon.jpg bufval",
    widget = "field_bmp",
    COMMENTS = "Press the arrow to enter a buffer value; F5 to check the value";
f004 = formonly.f_date type DATE,
    config = "icon.jpg bufval",
    widget = "field_bmp",
    COMMENTS = "Press the arrow to enter a value from the buffer";

f005 = formonly.thanx, config = " ", widget = "label",
    COLOR = RED;
```



```
f001 = formonly.tip,  
    config = "Click arrow buttons to fill values from the buffer",  
    widget = "label", COLOR = GREEN BOLD, center;  
  
f006 = formonly.check_v, widget = "button", config = "F5 {Check Value}", COLOR =  
YELLOW;  
f007 = formonly.qit, config = "qit {QUIT}",  
    widget = "button", COLOR = RED;  
f008 = formonly.ent, config = "return {Enter}",  
    widget = "button", COLOR = MAGENTA;
```

The Image File

The function field widgets of the 'master_form' form need a button picture to be specified. We used the icon.jpg picture for this purpose. You can save the picture given below and use it in your program:



However, if you leave the source code as it is and do not include the picture with this name into your project, the default picture will be used. The same will happen if the picture you specify for the function field is not found at runtime.



Form Widgets for Restricted Input

This chapter introduces three dynamic form widgets which have not been mentioned before: the radio button and the checkbox field widget. These widgets may facilitate the process of input by giving the user a set of predefined answers from which they can select rather than prompting them to enter some value.

Such behaviour may be desirable, when the expected input must be limited to a number of options and must not include any other values, or when the input should be formatted. For example, if you want the user to specify their sex (male or female), it is advisable to use either a combo box or a radio button, because, if a usual field is used instead, the user may be not sure about the format in which the input should be done. There are a number of possibilities, they can write "male" or "female", "m" or "f", "man" or "woman", etc. But to process the data correctly the program might need a predictable result. If the combo box is used instead, the user will select the correct answer from the drop down list and any ambiguity will be eliminated.

This chapter will also deal with a number of built-in functions which are mostly applied at runtime and can make your application more flexible and user friendly. For example, they can populate or clean the drop down list dynamically depending on the situation.

The Radio Button Widget

The purpose of the radio button widget is to provide a selection group from which only one option can be chosen. Each option has its own value. When the input is performed from a radio button widget, the variable receives the value depending on the choice.

A radio button must be added to the SCREEN section and declared in the ATTRIBUTE section of the form specification file by following this syntax:

```
Field_tag = database.field_name,  
          config = "option1_value {option1 label}  
                    [option2_value {option2 label}...]",  
          widget="radio",  
          [class="key"],  
          [default="default value"];
```

Here *field_tag* represents the corresponding field tag in the Screen section of the form specification file, and *field_name* stands for the name of the field which follows the table name or the "formonly" keyword, e.g. f01 = formonly.mar_status.

The *widget* attribute sets the widget type to "radio" converting a normal field into a radio button widget. The *class* attribute is an optional attribute which specifies the way the program treats the user input. The absence of this attribute makes the program return a character value from the radio button widget. If this attribute is specified, the program regards the user's choice as an action or a keypress. The *default* attribute specifies the default value of the widget. This attribute has been earlier discussed in the [Input field attributes](#) section of the *Performing input* chapter. Here is a simple example of a radio button widget declaration.

```
DATABASE formonly  
Screen {  
What is your marital status? [f01] }
```



```
}
```

ATTRIBUTES

```
f01 = formonly.mar_status TYPE CHAR,  
      config = "married {Married} single {Single}",  
      widget="radio";
```

INSTRUCTIONS

```
DELIMITERS "[ ]"
```

This will create a radio button group which includes two options: "married" and "single". When the user picks one of these options, its value will be passed to the corresponding variable in the source file.

Defining the Options and Their Values

The "config" attribute includes a number of strings separated by spaces. These are the option labels and values assigned to them.

- The string without the braces sets the value for the first option. This value will be assigned to the variable, if this option is selected by the user during the input.
- The string in the braces assigns a label that will be displayed next to the first option of the radio button. The curly braces are optional.

The second set of strings will assign a value with an accompanying label to the second option, and so on. There can be as many strings as many items you want your radio button group to include. The succession of strings must be enclosed in quotes.

For example, in the following source code a CHAR variable "occupation" is declared.

```
DEFINE occupation CHAR (20)  
INPUT BY NAME occupation
```

In the ATTRIBUTES section of the form specification file we create a radio button group which includes two options.

```
f1 = formonly.occupation,  
      config = "Student {Student} Worker {Worker} Teacher {Teacher}",  
      widget = "radio";
```

If the user selects the button "Worker", the corresponding value "Worker" will be sent to the variable "occupation" defined in the source file. The same variable will receive the value "Teacher" if this option is picked.

In the next example, the value "30" will be sent to the INT data type variable "age" if the user selects the first option, and correspondingly "40" or "50" if these options are picked. If the user does not select anything and presses the Accept key, "30" will be assigned to the variable, because it is the default value and it is already selected when the input begins.

```
#4GL file:  
MAIN  
DEFINE age INT  
OPEN WINDOW win1 AT 2, 2 WITH FORM "pers_info"  
INPUT BY NAME age  
DISPLAY age at 2,2  
END MAIN
```



Here is the "pers_info" form:

```
DATABASE formonly
Screen {
    How old are you? [f01]
}

ATTRIBUTES
f01 = formonly.age TYPE INT,
    config = "30 {Thirty years old} 40 {Forty years old} 50 {Fifty years
old}",
    widget= "radio",
    default="30";

INSTRUCTIONS
DELIMITERS "[]"
```

When you run the application, you will see the following in the screen:

```
How old are you?  Thirty years old
 Forty years old
 Fifty years old
```



Note: It is advisable that you leave some blank lines below the radio button widget specification. The number of the blank lines should at least correspond to the number of the widget options. Otherwise, the options will probably not fit the screen or overlap/be overlapped with other form objects.

Triggering Actions

Instead of sending values directly to variables in a source file, a radio button can trigger a key or action event. A different key or action event will be triggered depending on the option selected, if you choose this method. In this case the class attribute should be added to the specification of the corresponding radio button: `class = "key"`. At the same time the `config` option should include the key names rather than values for each option. In the example below the configuration string comprises several keys that will generate certain events or actions when the user selects the corresponding buttons.

```
f01 = formonly.answer TYPE CHAR,
    config = "F1 {Accept} F2 {Decline}",
    widget = "radio",
    class= "key";
```

This code will invoke either the ON KEY (F1) or ON KEY (F2) clause of the input depending on which option the user chooses.

The radio button widget is deactivated by default. If you do not activate it in the source code, the user won't be able to choose from the widget options. You can activate the radio button set by displaying "!" to this field (you can also disable it by displaying "*" to it, just the same as with the [Button](#) widget). The DISPLAY statement should be placed in the BEFORE INPUT clause of the INPUT ... FROM `radio_button` statement:

```
MAIN
```



```
DEFINE answer CHAR (30), num INT
OPEN WINDOW win1 AT 2,2 WITH FORM "pers_info"
INPUT BY NAME answer
    BEFORE INPUT
        DISPLAY "!" TO formonly.answer
    ON KEY (F1)
        DISPLAY "The offer has been accepted" AT 2, 2
    ON KEY (F2)
        DISPLAY "The offer has been declined" AT 2, 2
END INPUT
END MAIN
```

The same attribute class option "key" is used to define some actions performed when a certain item in a radio button group is selected without the Accept button being pressed:

```
f01 = formonly.answer TYPE CHAR,
    config = "Send_mail {Send mail} No_mail {Don't send mail}",
    widget = "radio",
    class = "key";
```

If the user clicks on one of the options in the radio button group declared above, the corresponding ON ACTION ("Send_mail") or ON ACTION ("No_mail") clause will be executed.

```
MAIN
DEFINE answer CHAR (30)
OPTIONS
PROMPT LINE FIRST
OPEN WINDOW win1 AT 2,2 WITH FORM "111"
INPUT BY NAME answer
    BEFORE INPUT
        DISPLAY "!" TO formonly.answer
    ON ACTION (Send_mail)
        DISPLAY "Type your message in the text area" AT 2, 2
    ON ACTION (No_mail)
        PROMPT "Choose another way to contact us" FOR answer
END INPUT
END MAIN
```

The Checkbox Widget

The checkbox widget is used to make a choice between two options. Input from each checkbox is made to a single variable. A checkbox can pass one of the two values to this variable - one value for the checked state and another one for the unchecked state. If you place more than one checkbox in the manner the radio button options are situated, you can create a multi-choice selection. However, you must not forget that each check box belongs to a separate widget and passes its value to a separate variable, so a number of checkboxes do not form a single group unlike the radio button options.

Checkboxes are declared in the ATTRIBUTES section of the form specification file using the following syntax:

```
Field_tag = field,
config="checked value unchecked value {label},
default = "default value"
widget="check",
[class="default"|"key"];
```



Here *field_tag* stands for the field tag in the Screen section of the form file and *field* - for the field identifier which includes the name of the database table or the "formonly" word if the form is not connected to any database, and the name declared for the field e.g. f01 = formonly.field_name.

The *widget* attribute with "check" value is required to set the type of widget to checkbox.

The *class* attribute is similar to that of the radio button widget; its specifics will be discussed later in this section.

Defining the Checked and Unchecked values

The *config* attribute sets two values separated by a space which the corresponding variable will receive when the user clicks on the box. The 4GL treats the values depending on their order in the *config* attribute: the first value is treated as the value passed to the variable when the widget is checked, the second is passed to the variable when the widget is unchecked. Here you can also specify a label for the checkbox within the curly braces - the label will be displayed to the right of the check box.

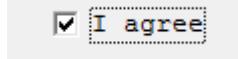
You should remember that if the checkbox widget is untouched during the input, the value sent to the variable depends on the default value. If there is no DEFAULT attribute, the unchecked value is sent to the variable. If the DEFAULT attribute is used in the widget declaration, the value specified as default is sent to the variable and the widget is displayed initially in the state specified by this attribute.

To specify the default value, include the "default" attribute in the form field specification:

```
default="checked_value" | "unchecked_value"
```

For example, when the program is run, the box labelled "I agree" in the code below will be checked at the beginning of the input. Otherwise it will be unchecked.

```
f1 = formonly.agreement,  
    config = "agree disagree {I agree}",  
    widget = "check",  
    default = "agree";
```



The line of code which follows will send the value "yes" to the corresponding variable if the user has checked the box, and "no" if the box is first checked and then unchecked or if it was left untouched, since no default value has been defined.

```
config = "yes no {I accept the conditions?}"
```



Note: When the box is checked, the value in the field changes. However, any logic related to it will be triggered only if complete the input. The widget can be checked or unchecked many times during the input, but the value assigned to the variable will be the value which was selected at the moment of completing the input.



Triggering Actions

Apart from writing a value to a particular variable directly, a checkbox can be used to generate key and action events. This may be necessary if you want the program to perform some actions at the moment the user checks or unchecks a box without the Accept key being pressed.

For this purpose you can add the attribute "class" which can be set to "default" or to "key":

```
class="default" | "key"
```

The "default" value is applied even if no "class" attribute is specified. It causes the checkbox to behave like a normal data checkbox returning values to the variable and not triggering any events, so specifying this attribute is not actually required.

If you specify "key" value, the checkbox will act as a button with two key or action events. As with the radio button widget, you have to display "!" to the checkbox in order to activate it.

The syntax for the default checkbox configuration string has been described above. As for a checkbox with the class attribute specified as "key", the configuration string includes the event to be triggered when the box is checked, the event when the box is unchecked instead of the values to be sent to the variable.

```
config="checked_key|checked_action unchecked_key|unchecked_action  
{label}"
```

The example below sends the key press F1 when the user ticks the box, and F2 if the box is not checked.

```
config = "F1 F2 {Do you agree?},class = "key";
```

The following example assigns action "act_print_text" to be performed when the box is checked, otherwise the action "act_print_html" will be fired.

```
f01 = formonly.nextAction,  
      config="act_print_text  act_print_html {Next Action}",  
      widget="check",  
      class="key";
```

Pay attention that you need to include the ON KEY and ON ACTION clauses into your 4GL source code file in the INPUT section to assign the behaviour for these keys or actions.

Functions for Working with Form Fields

Q4GL also offers some functions for working with any types of form widgets, such as retrieving information about the active field, changing the position of the cursor in the field, and others. Some of these functions are described further.

Finding the Position of the Cursor

There is a function which returns the position of the cursor in the currently active field. This is the *fgl_dialog_getcursor()* function. It accepts no arguments and can be invoked by a specified key or button.

In the sample code below when the cursor is in the field "firstname" and the F4 key is pressed, the user will see an integer that stands for the cursor position in this field.



```
#The "form1.per" form file:  
  
Database formonly  
Screen {  
  
    Enter your first name: [f01] ]  
}  
ATTRIBUTES  
f01 = formonly.firstname TYPE CHAR;  
INSTRUCTIONS  
DELIMITERS "[ ]"
```

The source file referencing the "form1.per" file is as follows:

```
MAIN  
DEFINE num INT, firstname CHAR (30), pos SMALLINT  
OPEN WINDOW win1 AT 2,2 WITH FORM "form1"  
INPUT BY NAME firstname  
ON KEY (F4)  
LET pos = fgl_dialog_getcursor ()  
DISPLAY pos AT 3,2  
LET num = fgl_getkey()  
END INPUT  
CLOSE WINDOW win1  
END MAIN
```



Note: Don't forget to include the END INPUT keywords after you have specified the key button for calling the *fgl_dialog_getcursor()* function.

Finding the Name of the Active Field

If you need to know the name of the currently active field, that is the field where the cursor is positioned at the moment, you can use the *fgl_dialog_getfieldname()* function. It requires no arguments and returns the character value representing the name of the field in which the cursor is located.

The example below illustrates the application of the function. When the cursor is in the fields "firstname" or "lastname", and the user presses the F4 key, the name of the related field will be displayed to the screen.

```
MAIN  
DEFINE num INT, firstname, lastname CHAR (30), fld_name CHAR (10)  
OPEN WINDOW win1 AT 2,2 WITH FORM "form1"  
INPUT BY NAME firstname, lastname  
ON KEY (F4)  
LET fld_name = fgl_dialog_getfieldname ()  
DISPLAY "The name of the active field is " , fld_name at 3,2  
LET num = fgl_getkey()  
END INPUT  
END MAIN
```



Moving the Cursor to a Definite Position

Q4GL supports a function which moves the cursor in the currently active field to a specific position. It is the *fgl_dialog_setcursor()* function

The function accepts a single argument – an integer denoting the position (in characters) within the active field where the cursor should be moved.

```
fgl_dialog_setcursor(cursor_position)
```

If you assign "1" or "0" as the argument value, the whole contents of the current field will be automatically highlighted. If this value exceeds the length of the defined variable, the cursor will be positioned after the last element in the field.

In the code below the cursor will be moved to the third character within the fields "firstname" or "lastname" when they are made active and the user presses the F2 key.

```
MAIN
DEFINE firstname, lastname CHAR (20)
OPEN WINDOW win1 AT 2,2 WITH FORM "form1"
INPUT BY NAME firstname, lastname
ON KEY (F4)
CALL fgl_dialog_setcursor(3)
END INPUT
END MAIN
```

Getting Information About the Current Field

You can use the *fgl_formfield_getoption()* function to retrieve some information about the currently active field (its column, line positions or its length).

The function accepts one argument - a CHAR data type variable which can be one of the following:

- X – The column position within the screen
- Y – The line position within the screen
- Length – The length of the field in character spaces

```
fgl_formfield_getoption(option)
```

The following example demonstrates the use of the *fgl_formfield_getoption()* function. When the user presses F5, F6 or F7 with the cursor located in the "firstname" or "lastname" fields, the information about these fields (their column and line positions or their length) will be displayed to the screen

```
MAIN
DEFINE num INT, firstname, lastname, info CHAR (30)
OPEN WINDOW win1 AT 2,2 WITH FORM "form1"
INPUT BY NAME firstname, lastname
BEFORE INPUT
ON KEY(F5)
LET info = fgl_formfield_getoption("x")
```



```
DISPLAY "X: ", info at 2,2
ON KEY (F6)
LET info = fgl_formfield_getoption("y")
display "Y: ", info at 3,2
ON KEY (F7)
LET info = fgl_formfield_getoption("length")
DISPLAY "Length: ", info at 4,2
END INPUT
LET num = fgl_getkey()
END MAIN
```

The screenshot shows a window with the following content:

```
X: 26
Y: 3
Length: 32

Enter your first name: 
Enter your last name: 
```

Script Options

The script options described in this section are applied in order to change some characteristics of a particular field or a group of fields, namely to change their border style and prevent their contents from being incidentally overwritten by a single key press.

Manipulating Field Borders

With the field-level script resource “border” you can set an individual style for a particular form field or a group of fields in a specific application, window or form.

A field may:

- have the 3D border
- have a flat border (a solid line around the field)
- have no border at all

Correspondingly, the “border” script resource has the options that affect the style of the border presentation.

By default all form fields are rendered with the 3D border like that in the following screenshot.

The screenshot shows a window with the following content:

```
Enter your first name: 
```

If you want to choose a different style, you should change the script option explicitly by including the “border” option followed by a colon and the desired style variant.



Here is the syntax:

```
application.window.form.table.field.border: border_style
```

For instance, the following code will change the border of the field called "firstname" in the form "my_form" opened in the window *my_win* of program *my_app*/from the 3D format to flat, which means there will appear a solid black line around this field. You can see it in the screenshot that follows.

```
my_appl.my_win.my_form.formonly.firstname.border: flat
```

The screenshot shows a simple Windows-style application window. At the top, the title bar reads 'my_form'. Below the title bar, there is a label 'Enter your first name:' followed by a rectangular input field. The input field has a thin black border, indicating it is currently selected or active.

If the border is not necessary, you should replace the word "flat" with "none":

```
my_appl.my_win.my_form.formonly.firstname.border: none
```

The screenshot shows the same application window as before, but the input field now has no border, appearing as a plain white rectangle.

When the border style should be changed for several fields in the same table, form, or window, the corresponding parts in the previous examples are replaced with question marks:

```
my_appl.my_win.my_form.?.firstname.border: flat
```

In this example the border for the field "firstname" which belongs to any table in the form *my_form* opened in window *my_win* when the program *my_app* is run will become flat.

And the following line of code will remove the border from any fields in any windows or forms of the program *my_app*.

```
my_appl.?.?.?.?.border: none
```

Field Overwrite Script

When the current field with some value entered in it is highlighted in the edit mode, i.e. when the cursor is positioned in it and the value is highlighted, a single key press will overwrite its contents, and the value entered last will be sent to the receiving variable instead of the value that the user intended to send there originally. To prevent this from happening you can use the *fgloverwrite* script option. It can accept two BOOLEAN values: TRUE and FALSE.

```
application.window.form.table.field.fglovewrite: BOOLEAN
```

TRUE is set by default, which means that when the cursor is moved to another field, this field becomes highlighted automatically.

The field called "firstname" in the example below will be highlighted the moment the cursor is placed there, and the user can start editing it without having to select its contents manually.



```
my_appl.win1.form1.formonly.firstname.fgloverwrite: true
```

The screenshot shows a user interface with two text input fields. The top field is labeled "Enter your first name:" and contains the text "Jack". The bottom field is labeled "Enter your last name:" and contains the text "Smith". Both fields have a blue border around them.

Then, as has already been said, a single key press will overwrite the contents of this field, and this value will be passed to the appropriate receiving variable.

If now we change the BOOLEAN value in the script to FALSE, the field will not be highlighted automatically, and the user will have to select its contents in order to alter the field value. This can be seen in the screenshot that follows.

```
my_appl.win1.form1.formonly.firstname.fgloverwrite: false
```

The screenshot shows a user interface with two text input fields. The top field is labeled "Enter your first name:" and contains the text "Jack". The bottom field is labeled "Enter your last name:" and contains the text "Smith". Both fields have a grey border around them.

Example

This application illustrates the form widgets, functions and script options described above in this chapter. The application name is 'restricted_input'. If you want the application to have another name, do not forget to change the application name parameter in the script file given below.

```
#####
# This example uses the radio buttons and check boxes to facilitate the input
# process and make the input values more predictable and thus easier to process.
#####
DEFINE
personal_data RECORD
  fullname CHAR (20),
  confirmed DATE,
  position CHAR (30),
  wage DECIMAL,
  gender CHAR,
  education CHAR (15),
languages RECORD
  eng, sp, ru VARCHAR(3)
END RECORD
END RECORD,
license CHAR,
experience INTERVAL YEAR TO MONTH,
```



```
ext_conf,tod  DATETIME YEAR TO MONTH,
curs_pos      INTEGER,
fname         CHAR(15),
op_x,op_y,
op_len        INTEGER,
answer        INTEGER,
getk          INTEGER

MAIN
# The first IF statement of the MAIN program block is used to check whether the
# application is run in GUI mode. If the Application is launched in character
# mode, a message will appear and the program will be terminated, because
# some of the form widgets cannot be used only in GUI mode.
IF fgl_fglgui() = FALSE
  THEN
    CALL fgl_winmessage("Character mode",
      "You tried to launch the program in character mode." ||
      "\nSome form widgets cannot be displayed and used correctly." ||
      "\nThe program will be terminated." ||
      "\n\nPlease, run the program in GUI mode", "error")
    EXIT PROGRAM
  END IF

OPTIONS FORM LINE 2 -- The form will be displayed at the second line
OPTIONS MESSAGE LINE 1 -- The messages will be displayed to the first line
OPEN WINDOW win1 AT 2,2 WITH FORM "personal_form"

# The following DISPLAY statements activate the form buttons
# and the license check box
DISPLAY "!" TO license
DISPLAY "!" TO curs_pos
DISPLAY "!" TO qit
DISPLAY "!" TO field_option
DISPLAY "!" TO save
MESSAGE ""

# Here we start the input of the personal data
INPUT personal_data FROM personal.*
AFTER FIELD confirmed -- after the user enters or chooses the value for
                      -- the confirmed field, the AFTER FIELD block
                      -- calculates how much time the employee has been
                      -- employed

# We use the EXTEND function to cast the entered date to the format necessary
# for the YEAR TO MONTH interval
LET ext_conf = EXTEND (personal_data.confirmed, YEAR TO MONTH)
LET tod = CURRENT -- the CURRENT value is automatically changed to the
                     -- YEAR TO MONTH format according to the tod variable
                     -- specification

LET experience = tod - ext_conf -- Calculates the time interval
DISPLAY experience TO exp -- Displays the time interval to the
                           -- exp field

ON KEY (CONTROL-Y) -- The key "Y" is associated with the license field.
                     -- When the user checks the license checkbox
```



```

INPUT license FROM lic_type -- the program activates a nested INPUT block
-- used to input the value to the
-- variable license

BEFORE INPUT
    DISPLAY "!" TO ok
ON KEY (CONTROL-N)          -- When the user unchecks the license checkbox,
    INITIALIZE license TO NULL -- the nested INPUT block is terminated and
    EXIT INPUT                -- the license variable is set to null

AFTER INPUT
    DISPLAY "*" TO ok
END INPUT

ON ACTION ("curs_pos")           -- the curs_pos (Cursor position) button
    LET curs_pos = fgl_dialog_getcursor() -- is used to display the current
                                         --position of the cursor in the field
    LET fname = fgl_dialog_getfieldname() -- is used to return the name of the
                                         -- current field
    CALL fgl_winmessage ("fgl_dialog_getcursor()",
    "Current field is: "||fname||
    "\nThe current position of the cursor is: "||curs_pos, "info")

ON ACTION ("field_option")       -- the field_option (Field Option) button
-- is used to display the current field
-- parameters:
    LET op_x = fgl_formfield_getoption("x") -- returns the column on which the
                                         -- field is situated
    LET op_y = fgl_formfield_getoption("y") -- returns the row on which the
                                         -- field is situated
    LET op_len = fgl_formfield_getoption("length") -- returns the length of the
                                         -- field
    LET fname = fgl_dialog_getfieldname() -- is used to return the name of the
                                         -- current field

# Displays the returned values in a message box
CALL fgl_winmessage("fgl_formfield_getoption",
    "The field "||fname CLIPPED||" is situated on:"||
    "\ncolumn: "||op_x||
    "\nline: "||op_y||
    "\nThe form field length is: "||op_len,
    "info")

ON KEY (F2)           -- the field_name (Field Name) button
    LET answer = fgl_message_box("Quit",
        "Do you want to quit?"||
        "\n(Press YES to exit the program)", 4)
    IF answer = 1 THEN EXIT PROGRAM END IF

AFTER INPUT
    LET answer = fgl_message_box("Save and exit",
        "Do you want to Save the inputted values and try again?"||
        "\n(Press YES to terminate the input)", 4)
    IF answer = 1 THEN
        CALL display_inp()
        CONTINUE INPUT -- prevents the input from automatic termination
    END IF

```



```

END INPUT
CALL fgl_getkey()
END MAIN

#####
# The function opens a new window with the form personal_form
# and displays the inputted values to this form.
#####
FUNCTION display_inp()
OPEN WINDOW win2 AT 2,2 WITH FORM "personal_form"
DISPLAY "!" TO save -- Activates the button save (F1)
DISPLAY "Back" TO save -- Changes the label of the button save (F1)
DISPLAY "!" TO qit -- activates the button qit (F2)
DISPLAY personal_data TO personal.*
DISPLAY experience TO exp
DISPLAY license TO lic_type
LABEL retry: -- The labelled place is used to give the user an opportunity
-- to chose whether to input new values or quit the program
LET getk = fgl_getkey() -- the function returns the number of the key pressed
IF getk = 343 THEN CLOSE WINDOW win2 ELSE -- 343 stands for the ACCEPT key
IF getk = 3001 THEN EXIT PROGRAM -- 3001 stands for the F2 key
ELSE GOTO retry END IF
END IF
END FUNCTION

```

The Form File

The following form file is named 'personal_form.per':

Database formonly

Screen

```

{
\gp-----[ L01 ]-----q\g
\g| Full Name:\g[ f01 ]-----|\g
\g|-----[ f02 ] \gExperience (yy-mm):\g [ f03 ]-----|\g
\g|-----[ f04 ]-----|\g
\g|-----[ f05 ]-----|\g
\g|-----[ f06 ]-----|\g
\g-----[ f07 ] \gLanguages:\g[ f08 ] \gLicence:\g[ f11 ][ f12 ]-----|\g
]|\g-----[ f09 ] [ b05 ]-----|\g
\g-----[ f10 ]-----|\g
\g-----[ b01 ] [ b02 ]-----[ b03 ] [ b04 ]-----|\g
\gb-----d\g

```



}

```

ATTRIBUTES
f01 = formonly.name, COLOR=BOLD, REQUIRED,
      COMMENTS = "Enter the first and the last name of the employee";
f02 = formonly.confirmed TYPE DATE, widget = "calendar", REQUIRED, AUTONEXT,
      COMMENTS=" Enter the date of confirmation";
f03 = formonly.exp TYPE INTERVAL YEAR TO MONTH;
f04 = formonly.pos, REQUIRED,
      COMMENTS = "Enter the current post title";
f05 = formonly.wage TYPE DECIMAL, DEFAULT = 1000.00, LEFT;
f06 = formonly.gender,
      config = "m {male} f {female}", DEFAULT = "male",
      widget = "radio";
f07 = formonly.educ,
      config = "higher {Higher} prof {Professional} second {Secondary}",
      DEFAULT = "Higher", widget = "radio";
f08 = formonly.lang_en,
      config = "yes no {English}",
      widget = "check";
f09 = formonly.lang_sp,
      config = "yes no {Spanish}",
      widget = "check";
f10 = formonly.lang_ru,
      config = "yes no {Russian}",
      widget = "check";
f11 = formonly.license,
      config = "CONTROL-Y CONTROL-N {}",
      widget = "check", class ="key";
f12 = formonly.lic_type,
      config = "A {A(Motorbike)} B {B(Motor car)} C {C(Truck)}",
      widget = "radio",
      COMMENTS= "Press Accept after you chose the license type";

b01 = formonly.curs_pos,
      config = "curs_pos {Cursor Position}", widget = "button";
b02 = formonly.field_option,
      config = "field_option {Field Options}", widget = "button";
b03 = formonly.save,
      config = "ACCEPT {Save}", widget = "button";
b04 = formonly.qit,
      config = "F2 {Quit}", widget = "button";
b05 = formonly.ok,
      config = "ACCEPT {OK}", widget = "button";

L01 = formonly.headr, config = "PERSONAL DATA", COLOR=RED BOLD,
      widget = "label", center;

INSTRUCTIONS
SCREEN RECORD personal (name, confirmed, pos, wage, gender, educ,
lang_en, lang_sp, lang_ru)

```

The Script File

Below is given the script file which changes the form fields options



```
#This script specifies the flat border for the name field
restricted_input.??.formonly.name.border: flat

#These script options specify 3D borders for the
#confirmed and exp form fields
restricted_input.??.formonly.confirmed.border: 3D
restricted_input.??.formonly.exp.border: 3D

#This script option removes the field border from all the
#fields of all the forms that will be opened by the program
#restricted_input
restricted_input.??.formonly.??.border: none

#This script option disables the automatic field overwriting
# for all the fields
restricted_input.??.formonly.??.fgloverwrite: false

#This script option enables the automatic field overwriting
#for the field named pos
restricted_input.??.formonly.wage.fgloverwrite: true

#The following options hide the buttons that won't be used
#during the inputted data display
restricted_input.win2.??.formonly.curs_pos.hidden: true
restricted_input.win2.??.formonly.field_name.hidden: true
restricted_input.win2.??.formonly.field_option.hidden: true
restricted_input.win2.??.formonly.ok.hidden: true
```





Input From a List

A graphical form can contain a widget which lets the user chose an input value from a drop down list or to enter their own value to the input field. Such widget is called a combo box.

The combo box widget is a field with a down arrow at the right end. Clicking on the arrow will display a drop-down list of values for selection. This list can be populated statically by including corresponding field attributes in the form specification file or dynamically at runtime by the user entering their own values which are not included in the combo box list.

Navigation within the list can be performed by using the mouse, the keyboard cursor keys or alpha numerical keys.



Note: When clicking on an item in the combo list the value in the field changes. However, any logic related to the field will not be triggered unless the input is completed.

Creating a Combo Box Widget

To create a combo box you should stick to the following syntax in the ATTRIBUTES section of the form specification file:

```
Field_tag = field,  
    widget="combo",  
    include = (list_item1, list_item2,... list_item_n),  
[class="combo"];
```

Here *tag* stands for the field tag in the Screen section of the form file and *field* - for the field identifier which includes the name of the database table or the "formonly" word, if the form is not connected to any database, e.g. f01 = formonly.field_name.

The *class* attribute is required only if you want to make the combo box editable: class = "combo". Without it the user will only be able to perform selection from the list. You should remember, however, that when you specify the class = "combo" attribute, the data in the include list will no longer be validated by the server.

The List of Values

The range of values that the drop-down list will contain is specified in the INCLUDE attribute. The syntax of the INCLUDE attribute is as follows:

```
INCLUDE = (value_list)
```

The *value list* can include a single value, a set of values separated by commas, and a *first TO last* structure.

Values can be represented by literal values, operators, built-in functions, TRUE, FALSE, and NOTFOUND constants and can also include the NULL value. The list of values cannot include a name of a variable or a programmer-defined function.



If you want the user to be able to leave the field empty without the `class="combo"` attribute specified, add the `NULL` value to the `value` list. When you apply the `INCLUDE` attribute to `formonly` fields, you must specify the field data type:

```
mar_stat = formonly.marital TYPE CHAR,  
          widget="combo",  
          INCLUDE =(NULL, "married", "single");
```

In this example, the user can choose two values in the field `mar_stat`: `married` and `single`, or can skip the data entry by pressing the RETURN key.

You can specify a range of values by using the `first` TO `last` structure which includes all the values between the first one and the last one. In this case, `first` value must be lower than `last` value, but it doesn't mean that these values must be of numeric data types only. You should remember these rules concerning different data types when you use the TO clause of the `INCLUDE` attribute:

- For the fields that support number and INTERVAL values, the `first` value is the positive number which is closer to zero, if both values are positive. If one or both values are negative, the `first` value is the larger (or only) negative value of the two specified.
- If the field supports date and time values, the `first` value is the one which indicates the earlier moment of time.
- If the field supports character values, the `first` value is the one starting with a character which is closer to the beginning of the ASCII character list. For example, such ranges as `A TO P`, `ab TO fr`, are valid. If you use digits, remember, that the range `7 TO 20`, which is valid in a numeric field, will be invalid in a character one, because in ASCII sequence, 7 (code number 55) comes after 2 (code number 50).



Note: The `INCLUDE` attribute doesn't treat a string of blank spaces as a `NULL` value, because the ASCII distinguishes between the blank spaces and the `NULL` value.

It is advisable, that you use the `COMMENTS` attribute together with the `INCLUDE` attribute in order to explain the user which values are expected for the current field:

```
integ = formonly.integ TYPE INT,  
  
COMMENTS = "You can input only integers here",  
  
INCLUDE = (NULL, 10 TO 99);
```



Note: By default normal fields with the `INCLUDE` attribute will be rendered as a combo box. This can be changed by means of the script resource `nocombo`.

The example below illustrates how a combo box is created.

```
DATABASE formonly
```



```
SCREEN {  
  
    Select your country: [f01]  
  
}  
  
ATTRIBUTES  
f01 = formonly.countries type CHAR,  
      widget="combo",  
      include = ("USA", "Canada", "UK"),  
      class = "combo";  
INSTRUCTIONS  
DELIMITERS "[ ]"
```



This code creates a combo box field with a list of values including the names of some countries. As the class property is specified (class = "combo"), the user is able to supply their own data. Remember that the attributes (*widget*, *class*, *include*) must be separated by commas.



Note: You must always specify a data type for formonly fields. If you don't, an error will be produced while trying to convert the combo box list data type to the field data.

Functions for Modifying the Drop Down List at Runtime

To provide the developer with an ability to manipulate entries in the combo box list at runtime, the corresponding built-in functions have been implemented in Q4GL. They make the process of the combo box list modification easy and fast. Some of these functions are explained below.

Clearing Items from the Combo Box List

There is a function which removes items from the combo list according to the index value associated with every item in the list for the specified field.

Function *fgl_list_clear()* accepts three arguments: *field_name*, *start_index* and *end_index*. The *field_name* argument is of the CHAR data type whereas both *start_index* and *end_index* are represented by integers.

The *start_index* argument defines the first item to be cleared from the list, and *end_index* indicates the position of the last item to be removed.

```
fgl_list_clear ("field_name", start_index, end_index)
```



Note: The "field_name" argument must be enclosed in double or single quotes, if it is not stored in a variable. Otherwise the operation of the function will terminate with an error. If there is a variable which name matches the name of the field, 4GL will treat the contents of this variable as the field name rather than its name.

If the function has executed successfully, it will return "1". In the event of failure "0" is returned.



Note: When you specify the "0" value for the *start_index* variable, all the items from the combo box list will be removed.

Below is an example of the *fgl_list_clear()* function application. The first two items from the box list in the previous example ("USA" and "Canada") are removed by calling the *fgl_list_clear()*. The only remaining entry is "UK".

"Combo_form.per":

```
SCREEN {
  [f01
  ]
ATTRIBUTES
f01 = formonly.countries type CHAR,
  widget="combo",
  include = ("USA", "Canada", "UK"),
  class = "combo";
INSTRUCTIONS
DELIMITERS "[ ]"
```

The source file:

```
MAIN

DEFINE num, start_index, end_index SMALLINT, countries CHAR (30)

OPEN WINDOW win1 AT 2,2 WITH FORM "combo_form"

INPUT BY NAME countries

LET start_index = 1

LET end_index = 2

CALL fgl_list_clear ("countries",start_index,end_index)

INPUT BY NAME countries
```



```
LET num = fgl_getkey()  
  
END MAIN
```



Counting the Number of Items in the Combo Box List

The *fgl_list_count()* function is used to return the number of items in the select list of a combo box. It takes a single argument – the name of a combo box field (char(255)), and returns a positive integer representing the number of list entries in this field.

```
fgl_list_count("field_name")
```



Note: The argument within the parentheses must be enclosed either in single or in double quotes, if you use the literal field name. The argument can also be represented by a character variable storing the name of the field.

Here is an example of how the *fgl_list_count()* function is used.

```
MAIN  
  
DEFINE num, num_items INT, countries CHAR (30)  
  
OPEN WINDOW win1 AT 2,2 WITH FORM "combo_form"  
  
INPUT BY NAME countries  
  
LET num_items = fgl_list_count ('countries')  
  
DISPLAY num_items AT 1,1  
  
LET num = fgl_getkey()  
  
END MAIN
```

As there are three elements in the combo-box list, the number that the user will see on the screen will also be "3".

Finding the Position of a Specific List Item

Function *fgl_list_find()* is used to determine the position of the specified value in the combo box list. It accepts two arguments: the *field_name* of the CHAR (255) data type and the value, also represented by a character data type.

```
fgl_list_find ("field_name", value)
```



The result of the function execution is a positive integer indicating the position of the specified value in the list. If the function fails to produce a result, it returns "0". Note that the name of the field must be enclosed in quotes.

The source code that follows will return "1", which is the current position of the item "USA" in the combo box list.

```
MAIN

DEFINE pos INT, countries, value CHAR (30)

OPEN WINDOW win1 AT 2,2 WITH FORM "combo_form"

INPUT BY NAME countries

LET value = "USA"

LET pos = fgl_list_find ('countries', value)

CALL fgl_getkey()

DISPLAY "The position of the USA item is: ", pos AT 2,2

CALL fgl_getkey()

END MAIN
```



Finding the Name of a Specific List Item

The *fgl_list_get()* function retrieves the value of an item from the combo box list according to the index value specified.

It accepts two arguments: *field_name* and *item_position*. The *field_name* attribute is of the CHAR data type, and the *item_position* is an integer representing the index variable of the item in question.

```
fgl_list_get ("field_name", item_position )
```

As the result of the *fgl_list_get()* function operation, you will see the character value from the combo box list whose position was indicated as the second argument of the function. If you specify here a number which exceeds the total number of entries in the list, a NULL string will be returned.

Here is an example of how *fgl_list_get()* works.

```
MAIN

DEFINE num, item_position INT, countries, value CHAR (30)
```



```
OPEN WINDOW win1 AT 2,2 WITH FORM "combo_form"

INPUT BY NAME countries

LET item_position = 3

LET value = fgl_list_get ('countries', item_position)

INPUT BY NAME countries

DISPLAY "The item value is ", value at 2,2

LET num = fgl_getkey()

END MAIN
```

The code above will display “UK” to the screen because this is the third item in the list and the second argument is “3”. It is not required to define a special variable for storing an item position. Instead, you can put a literal integer within the parentheses: `fgl_list_get ("country", 3)`



Adding Items to the List

If you need to add a new entry to the existing combo-box list items at runtime, you can use the `fgl_list_insert()` function.

It accepts three arguments: `field_name`, `index_value` and `item_value`.

The `field_name` is the name of the field in the form specification file with or without the “formonly” word or table title, and enclosed in double or single quotes. The `index_value` variable indicates the position of the new item in the combo box list. If you specify here the position which exceeds the number of items currently in the list, the new item will be added to the end of the list. For example, if there are three items in the list at the moment, and you specify “5” as the value of the `index_value` variable, the new item will become the fourth. The existing list the value whose position was indicated as this argument, will be moved down, and the new entry will be inserted before it.

The last function argument “`item_value`” is the actual value that you want to see in the selection list. It must be surrounded by quotes.

```
fgl_list_insert("field_name", index_value, "item_value")
```

`field_name` and `item_value` are CHAR data type variables while `index_value` is an integer.



	Note: New entries will be added to the list and seen only at runtime. The list configuration won't change permanently until you make the necessary changes in the "include" attribute of the combo box widget specification in the form file.
---	--

If the function execution is successful, it will return an integer which stands for the position where the new entry has been inserted. If not, "0" will be returned.

The code below will insert the item "Spain" into the second position of the combo list including three country names. The entry that used to be in that position ("Canada") will get shifted down.

MAIN

```
DEFINE num, index_value INT, countries, item_value CHAR (30)

OPEN WINDOW win1 AT 2,2 WITH FORM "combo_form"

INPUT BY NAME countries

LET index_value = 2

LET item_value = 'Spain'

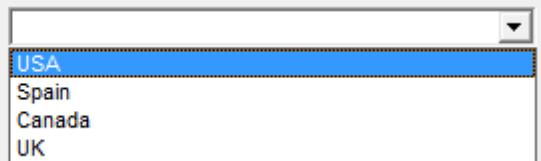
CALL fgl_list_insert('countries',index_value, item_value)

LET num = fgl_getkey()

INPUT BY NAME countries
LET num = fgl_getkey()

END MAIN
```

Select your country:



A screenshot of a Windows-style dropdown menu titled "Select your country:". The menu contains four items: "USA", "Spain", "Canada", and "UK". The "USA" item is highlighted with a blue selection bar.

It is not obligatory to create a special variable for storing an item value. You can put a literal enclosed in quotes right within the parentheses.

Replacing a Specific List Item

You can use the *fgl_list_set()* function to replace an existing combo box list value with another. It takes three arguments: *field_name*, *index_value*, and *new_value*.



As with the rest of the functions described in this section, the *field_name* variable denotes the name of a field in the form specification file with or without the "formonly" word or table title, and enclosed in single or double quotes. It is of the CHAR data type.

The second argument indicates the position of the list item that the user wants to be replaced. It is a positive integer. The last item in the argument list is the new value that will replace the value of the element in the specified position.

```
fgl_list_set("field_name", index_value, "new_value")
```

The *fgl_list_set()* function is similar to *fgl_list_insert()* in that it can add a new item to the combo box list if the second argument represents a value exceeding the total number of list entries. For instance, when for a list of three items the *index_value* in the *fgl_list_set()* function is set to "4", the new entry will simply be added to the list without replacing any existing values. So, both lines of code below will produce the same result.

```
CALL fgl_list_set ("country", 4,"Spain")
CALL fgl_list_insert("country", 4,"Spain")
```

If no errors occur, the *fgl_list_set()* function will replace an old list item value with a new one. Otherwise "0" will be generated.

Here is an example illustrating the use of the *fgl_list_set()* function. The first item in the existing list -"USA" is replaced by "Japan".

```
MAIN

DEFINE num, index_value INT, countries , new_value CHAR ( 30 )

OPEN WINDOW win1 AT 2,2 WITH FORM "combo_form"

INPUT BY NAME countries

LET num = fgl_getkey()

LET index_value = 1

LET new_value = "Japan"

CALL fgl_list_set('countries',index_value, new_value)

INPUT BY NAME countries

LET num = fgl_getkey()

END MAIN
```



Select your country:

The screenshot shows a standard Windows-style dropdown menu. The title bar says "Select your country:". Below it is a dropdown arrow. A list box contains three items: "Japan", "Canada", and "UK".



Note: New entries will replace old ones only at runtime. The list configuration won't change permanently until you make the necessary changes in the "include" attribute of the combo box widget in the form specification file.

Sorting the Combo Box List Entries

To sort the items in the list in a definite order, you can use the *fgl_list_sort()* function. It takes two arguments: *field_name* and *sort_index*.

The first argument represents the name of a field in the form specification file. It may or may not be preceded by the "formonly" word or the table name, but it must be enclosed in quotes. The field name can also be assigned to a variable of a character data type.

The second argument is an integer indicating the order in which the list elements should be sorted. If it is greater than or equal to "1", sorting will be performed in the ascending order, i.e. from 1 to 9 for integer values, for example, or from A to Z for character values.

If you specify the value of this variable as "0" or a negative number, the items will be sorted in the opposite, descending order (from 9 to 1 and from Z to A).

```
fgl_list_sort ("field_name", sort_index)
```

If no errors have occurred during the function operation, "1" is returned. Otherwise, the program will produce "0".

This screenshot shows the list before the function is invoked.

The screenshot shows a standard Windows-style dropdown menu. The title bar says "Select your country:". Below it is a dropdown arrow. A list box contains three items: "USA", "Canada", and "UK".

After the following code is executed, the names of the countries will be sorted in the ascending order as the *sort_index* variable value is represented by an integer greater than "1". You can see it in the screenshot after the example.

```
MAIN

DEFINE num, sort_index INT, countries CHAR (30)

OPEN WINDOW win1 AT 2,2 WITH FORM "combo_form"

INPUT BY NAME countries
```



```
LET sort_index = 2
CALL fgl_list_sort('countries',sort_index)
LET num = fgl_getkey()
INPUT BY NAME countries
LET num = fgl_getkey()
END MAIN
```



You don't have to define a separate variable for storing the *sort_index* value, and use a literal integer as this argument value instead: *fgl_list_sort ("country", 3)*.

Restoring the Original List

The *fgl_list_restore()* function is used to restore the initial configuration of the combo box list after it has been altered in some way by the built-in functions (certain items cleared from the list, inserted, modified etc.).

It accepts only one argument – the combo box field name optionally preceded by the "formonly" word or by the name of a database table. However, you must remember to enclose this argument value in quotes. Otherwise an error will occur.

```
fgl_list_restore ("field_name")
```

If the function is executed successfully, it returns "1", if there is an error, "0" is generated.

In the following example we first clear two entries from the list that comprises three country names. Then we call the *fgl_list_restore()* function and after it is executed, the list will have the same look as at the beginning of the program execution. You can follow these changes by pressing any key after each function call.

```
MAIN
DEFINE num INT, countries CHAR (30)
OPEN WINDOW win1 AT 2,2 WITH FORM "combo_form"
INPUT BY NAME countries
CALL fgl_list_clear ("countries", 1,2)
LET num = fgl_getkey()
```



```
INPUT BY NAME countries

CALL fgl_list_restore ("countries")

let num = fgl_getkey()

INPUT BY NAME countries

END MAIN
```

Example

This example illustrates how combo box widgets can be manipulated in all possible ways.

```
#####
# This example illustrates how to create combo boxes and manipulate their list
# items
#####

MAIN

DEFINE
    country_name,
    field_name,
    answer_char,
    info          VARCHAR ( 20 ),
    answer_int,
    num_days,
    answer_num   SMALLINT,
    answer_date,
    date_var     DATE,
    sort_index   INT

# Combo boxes and buttons are not fully functional
# in character mode, so use the GUI mode
IF fgl_fglgui()= 0 THEN
    CALL fgl_winmessage( "Wrong mode",
        "Run this application in the GUI mode.", "info" )
    EXIT PROGRAM
END IF

LET sort_index = 0 -- Then we set the sorting index to 0, which means the items
                  -- will originally be sorted in the descending order

OPTIONS
    ACCEPT KEY F3,
    # We include this option so that if the user closes
    # the information window using Close(X) button
    # it is treated as the Accept button
    ON CLOSE APPLICATION KEY F3

OPEN WINDOW TRAVEL_AGENCY_FORM AT 2,2 WITH FORM "travel_ag_form"
    ATTRIBUTES ( BORDER, FORM LINE 1)
```



```
DISPLAY "!" TO acc_but
DISPLAY "!" TO quit_but
DISPLAY "!" TO sort_but
DISPLAY "!" TO remove_but
DISPLAY "!" TO pos_but
DISPLAY "!" TO name_but
DISPLAY "!" TO replace_but
DISPLAY "!" TO insert_but
DISPLAY "!" TO count_but
DISPLAY "!" TO restore_but

# We activate the combo box fields for input
INPUT BY NAME country_name, num_days, date_var

ON KEY (F2)
LET answer_char = fgl_winquestion("QUIT?",
    "Do you really want to quit?", "Yes", "Yes|No", "question", 1)
CASE
    WHEN answer_char MATCHES "Yes"
        EXIT INPUT
    WHEN answer_char MATCHES "No"
        CONTINUE INPUT
END CASE

# Before the user moves to a new form field, a function is called.
# It returns the name of the active field, that is the field
# where the cursor is positioned at the moment, and displays this name to the
# information field
BEFORE FIELD country_name, num_days, date_var
CALL infd()RETURNING field_name
DISPLAY BY NAME field_name
CLEAR req_info

# Then we specify actions to be triggered whenever the buttons on the form are
# pressed. The actions will be applied to the currently active field.

ON ACTION ("Remove_items")
LET answer_int = fgl_winprompt(1,2,
    "Enter the id (integer number) of the item,|||
    "to be removed from the list."||
    "\nEnter 0, if you want to remove all the items.", "", 5,1)
IF answer_int IS NOT NULL THEN
    CALL fgl_list_clear(infd(), -- this functions returns the field name
                        answer_int) -- the ID of the list item
                        -- entered by the user
END IF

ON ACTION ("Sort")

CASE sort_index
WHEN 0 -- This is the sort_index initial value. Items are sorted in
       -- the descending order
    CALL fgl_list_sort(infd(), -1)-- when the button is pressed
```



```
-- for the first time.
DISPLAY "Sort items (asc)" TO sort_but -- The sorting operation is
                                         -- performed by means of
LET sort_index = 1 -- the fgl_list_sort() function. The second
                     -- argument is a negative number in this case.
                     -- Then we change the label on the button and
                     -- set the sorting index to 1.
                     -- Now the user can sort items in the ascending
                     -- order

WHEN 1
  CALL fgl_list_sort(infd(),1)
  DISPLAY "Sort items (desc)" TO sort_but
  LET sort_index = 0 -- When items have been sorted in the ascending
                     -- order, the sort_index variable is set to 0,
                     -- the label is changed again, and the next
                     -- button press will sort items in the
                     -- descending order

END CASE

ON ACTION ("Replace") -- When this action is activated, the user can replace a
                      -- particular list item by specifying its position and
                      -- new value

# Get the name of the current field
CALL infd() RETURNING field_name
# We prepare different messages which will be used in the
# prompt window depending on the current field
CASE field_name
  WHEN "country_name"
    LET info = "Enter a country name:"
  WHEN "num_days"
    LET info = "Enter a number:"
  WHEN num_days
    LET info = "Enter a date(mm/dd/yyyy):"
END CASE

LET answer_int = fgl_winprompt(1,2,
                               "Enter the position of the item to be replaced:", "", 5,1)
IF answer_int IS NOT NULL THEN
  # Here we use the prepared message
  LET answer_char = fgl_winprompt(1,2, info, "", 20,0)
  IF answer_char IS NOT NULL THEN
    CALL fgl_list_set(field_name, -- the name of the current field
                      answer_int, -- the position ID in the list
                      answer_char)-- the new value
  END IF
END IF

# The next action helps the user determine the position of the item whose value
# they enter into the prompt field. If that value does not belong to the
# item list, or it is a number, the function will return "0".
# The operation is executed by means of the fgl_list_find() function
ON ACTION ("Position")

# Get the name of the current field
```



```
LET answer_char = fgl_winprompt(1,2, "Enter the list item value:", "", 20,0)
IF answer_char IS NOT NULL THEN
    LET answer_int = fgl_list_find(infd(), answer_char)
    DISPLAY answer_int TO req_info
END IF

ON ACTION ("Name")-- This helps find out the value of the item in the
-- user-specified position

LET answer_int = fgl_winprompt(1,2, "Enter the item position ID:", "", 5,1)
IF answer_int IS NOT NULL THEN
    # The infd() function is used as an argument
    LET answer_char = fgl_list_get(infd(), answer_int)
    DISPLAY answer_char TO req_info
END IF

ON ACTION ("Count")-- The fgl_list_count() function will return the number of
-- items currently included in the combo box list. This
-- number will be displayed to the information field

CALL fgl_list_count (infd()) RETURNING answer_int
DISPLAY answer_int TO req_info

ON ACTION ("Restore") -- This action restores the original configuration
-- of a combo box list

CALL fgl_list_restore(infd())

ON ACTION ("Insert") -- This will insert new items into a combo box list
-- in the user-specified position.
-- The operation is performed by the fgl_list_insert
-- function

LET answer_int = fgl_winprompt(1,2,
    "Enter the position where the item should be insereted:", "", 5,1)
IF answer_int IS NOT NULL THEN
    LET answer_char = fgl_winprompt(1,2, "Enter a country name:", "", 20,0)
    IF answer_char IS NOT NULL THEN
        CALL fgl_list_insert(infd(), answer_int, answer_char)
    END IF
END IF

AFTER INPUT

# After the submitted data has been validated, an information message will
# appear on the screen, and the user can return to input by pressing a button
OPEN WINDOW INFORMATION AT 10,6 WITH 12 ROWS, 60 COLUMNS ATTRIBUTE (BORDER)
DISPLAY "THANK YOU! HERE IS THE INFORMATION YOU HAVE PROVIDED:"
    AT 4,4 ATTRIBUTE (BLUE, BOLD, UNDERLINE)
DISPLAY "DESTINATION:" AT 5,15 ATTRIBUTE (GREEN,BOLD)
DISPLAY country_name AT 5,33 ATTRIBUTE (BOLD)
DISPLAY "FOR: " AT 6,15 ATTRIBUTE (GREEN,BOLD)
DISPLAY num_days, " DAYS" AT 6,33 ATTRIBUTE (BOLD)
DISPLAY "DATE:" AT 7,15 ATTRIBUTE (GREEN,BOLD)
```



```
DISPLAY date_var AT 7,33 ATTRIBUTE (BOLD)
DISPLAY "PRESS ANY KEY TO CONTINUE INPUT" AT 10,14
    ATTRIBUTE (BLUE, BOLD, UNDERLINE)
CALL fgl_getkey()
CLOSE WINDOW INFORMATION
CONTINUE INPUT

END INPUT

CLOSE WINDOW TRAVEL_AGENCY_FORM

END MAIN

#####
# This function returns the name of the active field and displays
# it to a form field, it also returns the name of the current field.
# This function is used as an argument in the majority of the list functions
# here in place of the field name.
#####
FUNCTION infd()
CASE
WHEN INFIELD("country_name")

    RETURN "country_name"
WHEN INFIELD ("num_days")

    RETURN "num_days"
WHEN INFIELD ("date_var")

    RETURN "date_var"
END CASE
END FUNCTION
```

The Form File

This is the form "travel_ag_form.per":
DATABASE formonly

SCREEN

```
{
\g-----
 [f01] [f02] [f03] [f04] [f05] [f06]
\g-----
 \gp---[b001]---q\g [f07] [f11] [f15] [f16] [f17]
 | [f08] | [f12] | [f13] | [f14]
 | [f09] | [f10] | [f11] | [f12] | [f13] | [f14]
 | [f10] | [f11] | [f12] | [f13] | [f14]
\gb-----d\g \gp-----[b002]----q\g [f15] [f16] [f17]
 | [f11] | [f12] | [f13] | [f14]
 | [f12] | [f13] | [f14] | [f15] | [f16] | [f17]
 | [f13] | [f14] | [f15] | [f16] | [f17]
 | [f14] | [f15] | [f16] | [f17]
\gb-----d\g
```



```
\g-----\g
  [f18]           ][f19]           ][f20]
} }
```

ATTRIBUTES

```
f01 = foronly.country_name_lab, widget="label", config="DESTINATION", CENTER,
COLOR = BLUE BOLD;
f02 = foronly.num_days_lab, widget="label", config="NUMBER OF DAYS", CENTER,
COLOR = BLUE BOLD;
f03 = foronly.date_lab, widget="label", config="DATE", CENTER, COLOR = BLUE
BOLD;
# This is the first combobox. Note that we have specified its type - VARCHAR,
# which is obligatory for foronly fields. Also, the class attribute has been
# included, so that the user will be able to enter values which the list doesn't
# contain. The entries in the include attribute are listed in the jumbled order,
# and can be sorted by the user at runtime
f04 = foronly.country_name TYPE VARCHAR, widget="combo",
      include=("Egypt", "Australia", "USA", "Sweden", "Peru"),
      class="combo", DEFAULT="Peru", COLOR=BLUE BOLD,
      COMMENTS = "Choose the destination";
# These comboboxes' lists are populated by using the FIRST TO LAST structure.
f05 = foronly.num_days TYPE SMALLINT, widget="combo",
      include=(1 TO 9), class="combo", COLOR=BLUE BOLD, REQUIRED,
      COMMENTS = "Choose how many people who are going to travel", DEFAULT = "9";
f06 = foronly.date_var TYPE DATE, widget="combo",
      include = ("01/01/2011" TO "01/07/2011"), COLOR=BLUE BOLD,
      COMMENTS = "Choose the date", DEFAULT = "01/01/2011";
f07 = foronly.insert_but, widget="button", config = "Insert {Insert items}",
      COLOR=BOLD;
f08 = foronly.remove_but, widget="button",
      config="Remove_items {Remove items}", COLOR = BOLD;
f09 = foronly.replace_but, widget="button", config = "Replace {Replace items}",
      COLOR=BOLD;
f10 = foronly.sort_but, widget="button", config="Sort {Sort items (desc)}",
      COLOR = BOLD;
f11 = foronly.field_name_lab, CENTER, widget="label", config="ACTIVE FIELD",
      COLOR = BLUE BOLD;
f12 = foronly.field_name, CENTER, COLOR=GREEN BOLD;
f13 = foronly.req_info_lab, CENTER, widget="label",
      config="REQUESTED INFORMATION", COLOR = BLUE BOLD;
f14 = foronly.req_info, CENTER, COLOR=GREEN BOLD;
f15 = foronly.count_but, widget="button", config = "Count {Number of items}",
      COLOR = BOLD;
f16 = foronly.pos_but, widget="button", config = "Position {Item position}",
      COLOR = BOLD;
f17 = foronly.name_but, widget="button", config = "Name {Item value}",
      COLOR = BOLD;
f18 = foronly.restore_but, widget = "button",
      config = "Restore {Restore the original list}", CENTER,COLOR = GREEN BOLD;
f19 = foronly.acc_but, widget="button", config="F3 Submit", CENTER,
      COLOR = GREEN BOLD REVERSE;
f20 = foronly.quit_but, widget="button", config="F2 Quit", CENTER,
```



```
COLOR = RED BOLD REVERSE;  
b001 = formonly.info_label_1, widget = "label", config = "MODIFY",  
      CENTER, COLOR = BLUE BOLD;  
b002 = formonly.info_label_2, widget = "label", config = "INFO", CENTER,  
      COLOR = BLUE BOLD;
```



Performing Input from a Screen Array

We have already discussed the input of values to separate form fields, to program and screen records. The input into a screen array needs a special statement and special knowledge about performing such input.

The INPUT ARRAY Statement

The INPUT ARRAY statement allows the user to enter data to a screen array and assign these data to the elements of a program [array of RECORD](#) data type.

The general syntax of the INPUT ARRAY statement is as follows:

```
INPUT ARRAY Binding clause [HELP number] [ATTRIBUTE clause]  
[Control Block]  
[END INPUT]
```

The clauses enclosed in square brackets are optional.

The INPUT ARRAY statement passes the values entered by the user into fields of a screen array to a program array. To use the INPUT ARRAY statement, you should make the following steps:

- Create a screen array within a form file and compile the form file.
- Declare an ARRAY of RECORD within one of the DEFINE statements.
- Open and display the created form.
- Use the INPUT ARRAY statement to pass the user-entered values to the program array.

When the program starts executing the INPUT ARRAY statement, it displays the default values, if any, to the screen array fields, moves the cursor to the first field, and waits for the user to input a value. When the user enters a value and presses the ACCEPT or RETURN key (or when the field with the AUTONEXT attribute gets the value of specified length), the program assigns the value to the corresponding member of the specified screen array and moves the cursor to the next field.

The INPUT ARRAY statement is executed in the current form. The program deactivates the form when the statement execution is completed.

The Binding Clause

The binding clause is used to create temporary bounds between member variables of a program array and member fields of a screen array. These bounds are necessary in order to let the program manipulate with the entered values. This is the only obligatory clause in this statement. The syntax of the binding clause is as follows:

```
INPUT ARRAY program_array [WITHOUT DEFAULTS] FROM screen-array.*
```

Before you use them in the INPUT ARRAY statement, the program array must be defined in the DEFINE statement, and the screen array must be specified in the INSTRUCTIONS section of the form specification file.



The number of the members in each program record comprising program array must match the number of fields in each row of the corresponding screen array. The program array must be the array of RECORD data type. The elements that comprise the program array of record can be of any data type, but they must be of the same or of compatible data types with the corresponding field of the screen form. The program checks the entered value against the data type of the array elements, not against the field data type.

The number of screen records that comprise the screen array influences the number of rows that will be displayed to the form at one time. The number of array records that will be stored in your program is determined by the size of the array of record.

If the size of a program array is bigger than the size of the displayed screen array, the user can scroll through the screen array by means of Next Page and Previous Page keys, which are F3 and F4, respectively. The default order in which the cursor moves through the screen record fields is determined by the order in which the fields are listed in the screen record declaration clause.

The simplest example of the INPUT ARRAY statement usage may look as follows:

```
DEFINE my_parr ARRAY[10] OF RECORD
    member1 CHAR(20),
    member2 INT,
    member3 DATE
END RECORD
...
INPUT my_parr FROM my_sarr.*
```

The WITHOUT DEFAULTS Keywords

If you don't use the WITHOUT DEFAULT keywords after the program array specification, the program will display the default values to the screen array. The default values cannot be assigned by the programmer. 4GL supplies default values depending on the field data types; they are as follows:

Field type	Default value
Character	Blank (ASCII 32)
Number	0
INTERVAI	0
MONEY	\$0.00
DATE	12/31/1899
DATETIME	1899-12-31 23:59:59:99999

The fields that have data types which are not listed in this table get NULL as the default value.

If you include the WITHOUT DEFAULTS keywords to the binding clause, the program doesn't display the default values to the form fields. However, it displays the *current* values of the variables. The WITHOUT DEFAULTS keywords can prove useful when the user needs an opportunity to check and to correct the values that were assigned to variables during the program execution. The keywords can be used in a context like this:

```
INPUT first_year WITHOUT DEFAULTS FROM general.*
```



The ATTRIBUTE Clause

The INPUT ARRAY ATTRIBUTE clause generally resembles the ATTRIBUTE clause of the DISPLAY statement. You can specify text colour and intensity attributes for the values that are entered by user. We have discussed the syntax of these attributes in [Chapter 3](#) (i.e. BOLD, DIM, RED, BLUE, INVISIBLE attributes etc.).

The attributes specified within the INPUT ARRAY statement temporary override the attributes specified in OPTIONS and OPEN WINDOW statements, but they are in effect only during the current form activation.

When the input is over or the form is deactivated in any other way, the previous attributes come into effect. The following line will make the entered values be displayed in green and bold font:

```
INPUT my_parr FROM my_sarr.* ATTRIBUTES (GREEN, BOLD)
```

CURRENT ROW DISPLAY

4GL supports a set of unique attributes for the ATTRIBUTE clause of the INPUT ARRAY statement. The CURRENT ROW DISPLAY attribute specifies the display attributes for the row to where the input is currently performed. When the cursor moves to another row, the row that was highlighted gets the attributes that were in effect before the input to this row.

The syntax of the attribute is as follows:

```
ATTRIBUTES (CURRENT ROW DISPLAY = "keyword [, keyword_2, ... ]")
```

Where *keyword* stands for the *colour* or *intensity* attribute keywords, except for the DIM, INVISIBLE, and NORMAL keywords, which cannot be specified here. If you specify several attributes after the CURRENT ROW DISPLAY keywords, they must be separated with commas. The program will produce a compile-time error, if you specify no keywords in quotes.

An ATTRIBUTE clause can contain display attributes which apply to the whole screen array and the CURRENT ROW DISPLAY attribute at the same time.

In the following example, the values entered by the user will be displayed in blue, but the current row will be displayed in white. As the attributes of the current row and general attributes are independent from each other, we need to use the BOLD keyword twice, to specify that we want all the rows including the current one to be bold. When the cursor moves apart from the highlighted row, it becomes displayed in blue, and the next row is highlighted with white font colour:

```
INPUT ARRAY prog_arr FROM scr_arr.*  
ATTRIBUTE (BLUE, BOLD, CURRENT ROW DISPLAY = "WHITE, BOLD")
```

COUNT

The COUNT attribute is used to specify the number of program array elements that contain data. This attribute can be used only within the ATTRIBUTE clause of the INPUT ARRAY statement and has the following syntax:

```
COUNT = number
```



Where *number* specifies the number of program array records that contain data, and can be represented by a positive literal integer or by an INT or SMALLINT variable which contains such number.

The following lines specify, that the number of screen records that will be displayed to the screen array is 3:

```
INPUT ARRAY prog_arr FROM scr_arr.*  
    ATTRIBUTE (COUNT=3)
```

The effect of the COUNT attribute is similar to that of the *set_count* function positioned before the INPUT ARRAY statement. This means, that the example above has the same effect as the following lines:

```
CALL SET_COUNT(3)  
INPUT ARRAY prog_arr FROM scr_arr.*
```

MAXCOUNT

The MAXCOUNT attribute can be used to specify the dynamic size of a screen array. This size can be less than the size declared in the INSTRUCTIONS section of the form specification file.

The MAXCOUNT attribute can be specified only within the ATTRIBUTE clause of the INPUT ARRAY statement. The syntax of the attribute is:

```
MAXCOUNT = number
```

The *number* stands for the number of screen array records that can display data. The *number* can be represented by a positive literal integer, or by an INT or SMALLINT variable that contains such value.

The following example shows how the MAXCOUNT attribute can be used together with the COUNT attribute.

```
INPUT ARRAY prog_arr FROM scr_arr  
    ATTRIBUTE (MAXCOUNT = 10, COUNT = 5)
```

Here, 10 indicates the dynamic size of the screen array to where the program array is displayed, and 5 indicates the number of program array records which contain data.

If you specify the MAXCOUNT *number* as less than 1 or greater than the declared size of the program array, the program will use the original program array size as the MAXCOUNT *number* identifier.

The HELP Clause

The HELP clause is used to specify the number of a help message that will be displayed, if the user presses the Help key, when the cursor is positioned in the screen array. The default Help key is CONTROL-W, but it can be changed by means of the HELP KEY clause of the OPTIONS statement.



Note: This HELP clause assigns the HELP message that is linked to the whole screen array. If you want to specify a HELP message for a particular field, use the [SHOW_HELP\(\)](#) function as it was described in previous chapter.

The INPUT ARRAY Control Block

The INPUT ARRAY control block is an optional block which contains one or more INPUT ARRAY control clauses and is used to control the execution of the INPUT ARRAY statement. Each control clause should be comprised of at least one statement and an activation clause which specifies the conditions of the statement execution. You can include different-purpose statements to the INPUT ARRAY control block. These items are given below:

- The BEFORE INPUT and AFTER INPUT clauses which specify statements that are to be executed before or after the execution of the INPUT ARRAY statement;
- BEFORE ROW and AFTER ROW clauses contain statements that are to be executed before or after the user enters values to all the fields of the current screen array row;
- BEFORE INSERT/DELETE and AFTER INSERT/DELETE clauses contain the statements that are to be executed before or after the user inserts a row to the screen array or deletes it;
- BEFORE FIELD and AFTER FIELD clauses specify the statements that are to be executed before or after the user enters a value to the specified field of the current row;
- The ON KEY clause contains the statements that are to be executed when the user presses the specified combination of keys.

The [BEFORE INPUT](#), [AFTER INPUT](#), [BEFORE FIELD](#), [AFTER FIELD](#), [ON KEY](#), [ON ACTION](#) clauses are similar to those used in the INPUT or DISPLAY ARRAY statements. Their syntax, purposes, and main rules of their application have already been discussed in chapter 17.

The INPUT ARRAY control block may also include such keywords as NEXT FIELD. The usage of the NEXT FIELD keywords within an INPUT ARRAY statement is the same as in the INPUT statement and was described in the ["Filed order"](#) section of chapter 17.

When you include a Control clause to your INPUT ARRAY statement, don't forget to close the statement with the END INPUT keywords.

The BEFORE ROW Clause

The BEFORE ROW clause can appear only within the INPUT ARRAY statement. In the BEFORE ROW and AFTER ROW keywords, the ROW means a screen record, which is graphically represented as a row of a screen array within a screen form. The clause is executed each time before the user is allowed to enter data to the next row.

The INPUT ARRAY statement can contain only one BEFORE ROW block which is executed if:

- The cursor is moved to a new line of the screen form;
- The Delete key is pressed;
- A lack of space causes the INSERT statement failure;
- The user presses the Interrupt or Quit keys and terminates the INSERT statement execution.



The example below will suspend the input for 1 second before letting the user enter values in each next row and display the message:

```
INPUT ARRAY pr_arr FROM scr_arr.*  
  
BEFORE ROW  
  
    DISPLAY "You can now enter values for the next row." AT 2,2  
    SLEEP 1  
    DISPLAY " " AT 2,2  
  
END INPUT
```

The AFTER ROW Clause

The AFTER ROW block specifies the actions to be taken by the program every time after the user finishes the input to a row and leaves it.

One INPUT ARRAY statement may contain only one AFTER ROW block which is activated when the cursor moves to another row. The cursor can be moved to another row using the following keys:

- RETURN, TAB, or Accept key
- Any arrow key
- the Interrupt key (if the DEFER INTERRUPT statement was specified and executed earlier in the program)
- The Insert key (to add rows to a new screen record).

The CONTINUE INPUT Keywords

The CONTINUE INPUT statement directs the program to ignore all the statements that are located between the CONTINUE INPUT and END INPUT keywords and places the cursor to the field it occupied the last in the current form.

It is convenient to use the CONTINUE INPUT statement when the program control becomes nested deeply in a number of conditional loops and you need to return it to the user. The statement may also prove useful when applied within an AFTER INPUT control block designed to examine the field buffers. If their values need changing or correction, you can return the cursor to the form.

The EXIT INPUT Keywords

The EXIT INPUT statement is used to terminate the input process. When the program runs across the EXIT INPUT statement, it skips all the statements between EXIT INPUT and END INPUT, deactivates the form and passes the program control to the statement which is specified just after the END INPUT keywords.

If the EXIT INPUT statement comes in effect, the program ignores an AFTER INPUT block, if any.

Inserting and Deleting Rows

When the user performs the input into a screen array, they may need to add a row between the rows which already have their values. For example, if the two adjacent rows have the following values:



```
...
[5] [fifth row    ]
[7] [seventh row  ]
...
...
```

The user may want to add a row between these two rows with values "6" and "sixth row" not to violate the order of values. There is a way to do that without the need to re-input values for all the subsequent rows.

When the cursor is located in one of the fields of the fifth row, the user may press the Insert key to shift all the values located beneath it down one row and to free the space for adding another row of values. The Insert key is **F1** by default. This key can be changed using the OPTIONS statement as follows:

```
OPTIONS INSERT KEY <key_name>
```

The user can also delete all the values from a screen array row using the Delete key. In such case all the subsequent rows will be shifted up one row. The Delete key is **F2** by default. This key can be changed using the OPTIONS statement as follows:

```
OPTIONS DELETE KEY <key_name>
```

It is important to remember that inserting and deleting activities affect only the values in the screen array rows, not the physical appearance of the screen array. The Insert and Delete keys are unable to draw another row in the displayed screen array for the user to input additional values or delete one of the already displayed rows. The size of the screen array declared in the INSTRUCTIONS section of the screen specification file cannot be changed by these keys. So pressing an Insert key when all the rows of the screen array have values will not cause another row to appear, because the values have no free fields beneath to be shifted to. Pressing the Delete key only deletes the values from the row and not the row itself, thus, if there are no rows with values underneath the deleted row, it will remain empty and no values will be shifted to it.

Enabling and Disabling the Insert and Delete Keys

You can enable or disable the Insert and Delete keys with the help of some special attributes of the ATTRIBUTE clause of the INPUT ARRAY statement. By default the user is able to insert and delete rows while performing input.

The INSERT ROW Attribute

The INSERT ROW attribute of the ATTRIBUTE clause provides the programmer with an opportunity to enable or disable the Insert key (which is F1 by default) for the entire form when the INPUT array statement is being executed.

The INSERT ROW attribute can be followed by the TRUE or FALSE keywords. The syntax of the attribute is as follows:

```
INSERT ROW = TRUE | FALSE
```

The default specification for the INSERT ROW attribute is TRUE. When the INSERT ROW attribute is specified as TRUE, the user is allowed to use the Insert key when entering data.



When the programmer assigns the FALSE value to the INSERT ROW attribute, the Insert key is disabled, and the user cannot insert data during the INPUT ARRAY statement execution. However, the program still allows the user to insert values using the RETURN, TAB, and ARROW keys.

```
INPUT pro_arr FROM scr_arr.*  
ATTRIBUTE(INSERT ROW = FALSE)
```

When the user inserts a record, they must execute the following actions:

- Enter data to all the required fields of the current record;
- Use the RETURN, TAB, HOME, END, Accept, or any arrow keys to position the cursor to a field that is out of the current screen record.

The DELETE ROW Attribute

The DELETE ROW attribute of the ATTRIBUTE clause is used to enable or disable the Delete key (which is F2 by default) for the entire form when the user performs data entry.

As with the INSERT ROW, the DELETE ROW attribute can be followed by TRUE and FALSE keywords:

```
DELETE ROW = TRUE | FALSE
```

The default value for the DELETE ROW attribute is TRUE. When the DELETE ROW is specified as TRUE, the user is allowed to use the Delete key during the data input to the screen array to delete a row from the array. When the DELETE ROW is specified as FALSE, the Delete key is disabled and the user cannot use it during the data input.

```
INPUT pro_arr FROM scr_arr.*  
ATTRIBUTE(delete ROW = FALSE)
```

Control Clauses Influencing Inserting and Deleting of Rows

The INPUT ARRAY Control block can include special control clauses which are used to control the behaviour of the program when a row is deleted from an array or inserted into it. They are used together with other control clauses and also require the END INPUT keywords at the end of the INPUT ARRAY statement, if at least one of them is included.

The BEFORE DELETE Clause

The BEFORE DELETE clause specifies the actions that are to be executed after the user presses the Delete key, but before the array record is actually deleted and the values are shifted upwards. The INPUT ARRAY statement can contain only one BEFORE DELETE block which is executed each time the Delete key is pressed.

If you don't want the user to be able to delete array records, you can add the EXIT INPUT keywords within the BEFORE DELETE block. You can make the program cancel the user's delete operations by adding the CANCEL DELETE clause to the BEFORE DELETE clause. The syntax of the CANCEL DELETE keywords within the BEFORE DELETE control block is as follows:

```
INPUT pro_arr FROM scr_arr.*
```



```
BEFORE DELETE
    ERROR "You cannot delete rows"
    CANCEL DELETE
    ...
END INPUT
```

If the Delete operation was cancelled, it will have no effect on the rows that are processed during the current INPUT ARRAY statement execution.

If you use the CANCEL DELETE keywords outside the BEFORE DELETE control block, the program will produce an error or will ignore the keywords during the execution.

The AFTER DELETE Clause

The AFTER DELETE block specifies the actions to be performed by the program after the user clears a screen record by pressing the Delete key. If the programmer specifies the AFTER DELETE block and the user presses the Delete key, the program deletes the record from the screen array, executes the statements specified in the AFTER DELETE block, and then executes the statements in the AFTER ROW block, if any.

```
INPUT pro_arr FROM scr_arr.*
AFTER DELETE
    MESSAGE "You have deleted the row"
    ...
END INPUT
```

If the user wants the program to accept the changes that have been made, he or she must press the Accept key. One INPUT statement can contain only one AFTER DELETE block.

The BEFORE INSERT Clause

The BEFORE INSERT block is used to specify actions that will be performed before the user inserts a record into a screen array by means of the Insert key.

The program executes the statements that comprise the BEFORE INSERT block if one or several of the following conditions are met:

- The user starts to enter new records into the array;
- The user moves the cursor to a blank record placed at the end of the array;
- The user wants to insert a record between the two ones that are already input, and presses the INSERT key. In this case the BEFORE INSERT block is executed before the program adds a record to the array.

One INPUT ARRAY statement may contain only one BEFORE INSERT block.

You can make the program cancel the user's insert actions automatically by adding the CANCEL INSERT keywords to the BEFORE INSERT block. If the program has cancelled the insert operation, it won't have any effect on the rows that are processed by the current INPUT ARRAY statement. If the program comes across



the CANCEL INSERT keywords, the user won't be allowed to use the Insert, TAB, RETURN, ENTER, or ARROW keys to enter new rows.

```
INPUT pro_arr FROM scr_arr.*  
BEFORE INSERT  
    CANCEL INSERT  
    ...  
END INPUT
```

When the CANCEL INSERT or CANCEL DELETE keywords are executed, the program terminates the current BEFORE INSERT or BEFORE DELETE control block and the program control passes to the statement that is next to the terminated block.

The AFTER INSERT Clause

The AFTER INSERT block is used to specify actions that will be performed after the user inserts a record into the screen array. The AFTER INSERT block will have an effect only if a screen array is referenced by the FROM or BY NAME clause.

The AFTER INSERT block can't be activated by the Insert key itself. The program starts executing the block only as soon as the cursor is moved away from the record that has just been inserted.

You can include only one AFTER INSERT block to an INPUT ARRAY statement.

Screen Arrays with Graphical Widgets

The screen arrays discussed earlier could display and take only text data. However, graphical form widgets expand the input abilities and can make the input process easier. You can specify graphical form widgets as elements of screen array. For example:

```
DATABASE formonly  
  
SCREEN{  
  
    NAME      Married      Language     Country      Date  
    [txt]      ] [checkbox]  ] [radio]     ] [combo]     ] [calendar]  ]  
    [txt]      ] [checkbox]  ] [radio]     ] [combo]     ] [calendar]  ]  
    [txt]      ] [checkbox]  ] [radio]     ] [combo]     ] [calendar]  ]  
    [txt]      ] [checkbox]  ] [radio]     ] [combo]     ] [calendar]  ]  
    [txt]      ] [checkbox]  ] [radio]     ] [combo]     ] [calendar]  ]  
  
}  
  
ATTRIBUTES
```



```
txt = formonly.txt;
checkbox = formonly.checkbox,
            config = "Yes No { }",
            widget = "check";
radio = formonly.radio,
        config = "Eng {English} Oth {Other}",
        widget = "radio";
combo = formonly.combo,
        widget = "combo",
        include = ("USA", "England", "France", "Germany", "Finalnd");
calendar = formonly.calendar,
        widget = "calendar";
INSTRUCTIONS
SCREEN RECORD warray [5] (formonly.txt, formonly.checkbox,
                           formonly.radio, formonly.combo, formonly.calendar)
```

To check how such screen array works, add the following lines to the 4GL file:

```
MAIN

DEFINE myarr ARRAY[5] OF RECORD
    txt CHAR (10),
    chk CHAR(10),
    rad CHAR (10),
    comb CHAR (10),
    dt date

END RECORD

OPEN FORM f1 FROM "my_formfile"
DISPLAY FORM f1

CALL set_count(5)
INPUT ARRAY myarr FROM warray.*
CALL fgl_getkey()
END MAIN
```

When you run the application, you'll see the following input form:



NAME	Married	Language	Country	Date
Jack	<input checked="" type="checkbox"/>	<input checked="" type="radio"/> English	England	10/11/2010
	<input type="checkbox"/>	<input type="radio"/> Other		
	<input type="checkbox"/>	<input type="radio"/> English		
	<input type="checkbox"/>	<input type="radio"/> Other		
	<input type="checkbox"/>	<input type="radio"/> English		
	<input type="checkbox"/>	<input type="radio"/> Other		
	<input type="checkbox"/>	<input type="radio"/> English		
	<input type="checkbox"/>	<input type="radio"/> Other		
	<input type="checkbox"/>	<input type="radio"/> English		
	<input type="checkbox"/>	<input type="radio"/> Other		

Using the Built-In Functions for Manipulating the Input

4GL supports a set of functions that can prove useful while performing the input into a screen array. These functions can be helpful in regulating and evaluating the input values as well as in estimating and controlling the scope of the input.

The arr_curr() Function

The *arr_curr()* function is used to return an integer value that corresponds to the number of a record member of a program array that is displayed in the current line of the screen array. The computer treats the line as the current one, if the cursor is positioned in it at the beginning of the BEFORE ROW or AFTER ROW clause.

The program assigns number 1 both to the first line of the screen array and the first row of the program array. The *arr_curr()* function doesn't need the CALL statement to be invoked.

The example below will display the message containing the number of the current row each time the cursor will move to another program array record and thus the current row number will change.

```

INPUT pro_arr FROM scr_arr.*
AFTER ROW
    DISPLAY "The number of the current row is ", arr_curr()
...
END INPUT

```

The *arr_curr()* function can be invoked as an argument of another function. If this is the case, the function argument will be represented by the number of the current record which belongs to the array referenced by the INPUT ARRAY or DISPLAY ARRAY statements.

You can use the *arr_curr()* function to make the FOR loop begin beyond the first line of array. To do it, you can set a variable to *arr_cur()* and use this variable as the *first* value in the FOR loop.



The arr_count() Function

The *arr_count()* function is used to determine the total number of program records that a program array stores at the moment. When you call the *set_count* function, you can specify an upper limit for the value that can be returned by the *arr_count()* function.

The value returned by the *arr_count()* function is a positive integer, that corresponds to the index of the last record of the screen array to which the cursor can be positioned. It is important, that the *arr_count()* function can include rows that don't contain any data. Such situation may happen if the user scrolls the cursor through more fields but don't fill them with values.

In the following example, the program displays the number of records that have already been entered:

```
CALL SET_COUNT(50)

INPUT ARRAY pers FROM scr_rec1.*

AFTER ROW

DISPLAY "You have entered ", arr_count(), " records" AT 2,2

END INPUT
```

If the program has called the *set_count* function explicitly, the value returned by the *arr_count()* function depends on the value of the argument of the *set_count* function and the highest value of the array index. The *arr_count()* function returns the greater of these two values.

The fgl_scr_size() Function

The *fgl_scr_size()* function is used to return an integer value which corresponds to the number of screen records in a current screen array which is specified as the argument of this function.

The syntax of the *fgl_scr_size()* function invocation is as follows:

```
Fgl_scr_size(array)
```

Here "array" is the name of the screen array declared in the INSTRUCTIONS section of a form specification file. The array identifier must be enclosed in quotation marks. You can replace the screen array identifier by a variable of a CHAR OR VARCHAR data type which contains such identifier.

The following example is a modified version of the previous one. This snippet of a source code displays the number of the records which were entered and the number of entries that can be seen in the screen at once:

```
CALL set_count(50)

INPUT ARRAY pers FROM scr_rec1.*

AFTER ROW

DISPLAY "You have entered ", arr_count(), " records. AT 2,2
DISPLAY "The screen contains ", fgl_scr_size("scr"), "records." AT 2,2
```



The scr_line() Function

The *scr_line()* function is used to return a positive integer that indicates the number of the current screen record in its screen array when the DISPLAY ARRAY or INPUT ARRAY statements are being executed.

The 4GL indicates the current screen record as the one which contains the cursor at the beginning of a BEFORE ROW or AFTER ROW clause. The first records of program and screen array are numbered 1. If the program array and the screen array are of different sizes, the *scr_line()* function and the *arr_curr()* function can return different values.

In this example, we have added a new line to the program used above, and now the user can see to which record he or she is performing the input:

```
CALL SET_COUNT(15)

INPUT ARRAY pers FROM scr_recl.*

AFTER ROW

DISPLAY "You have entered ", arr_count() at 2,2

BEFORE ROW

DISPLAY "Now you are entering values to record ", scr_line() at
3,2

END INPUT
```

The fgl_dialog_setcurrline() Function

You can use the *fgl_dialog_setcurr()* function displays a specified row of the program array to the specified line of the screen array.

The function needs two arguments that are to be separated with a comma. The syntax of the function invocation is:

```
CALL fgl_dialog_setcurr(screen_line, prog_line)
```

Where *screen_line* stands for the number of the line in the screen array; *prog_line* stands for the number of the row in a program array.

In the following example, the members of the screen array get some values before the INPUT ARRAY statement is activated. The user can see the program-assigned values when pressing the CONTROL-*number* keys, where *number* stands for the number of a program array record which is to be displayed to the form:

```
FOR i = 1 TO 10
LET my_parr[i].m1 = "Line ", i
LET my_parr[i].m2 = "Some text"
LET my_parr[i].m3 = "Other text"
END FOR

INPUT ARRAY my_parr FROM my_sarr.*
```



```
ATTRIBUTES (red, bold, current row display = "Cyan, reverse")  
  
ON KEY (Control-1) CALL fgl_dialog_setcurrline (1,1)  
ON KEY (Control-2) CALL fgl_dialog_setcurrline (2,2)  
ON KEY (Control-3) CALL fgl_dialog_setcurrline (3,3)  
ON KEY (Control-4) CALL fgl_dialog_setcurrline (4,4)  
ON KEY (Control-5) CALL fgl_dialog_setcurrline (5,5)  
ON KEY (Control-6) CALL fgl_dialog_setcurrline (6,6)
```



Example

This example allows the user to perform input into a program array of records from a standard array consisting of text fields as well as from an array consisting of different types of widgets.

```
#####
# This example illustrates the input performed from a screen array
# into a program array of records
#####

DEFINE -- We define two program arrays
  a_country_list ARRAY[15] OF RECORD
    country_id  CHAR(3),
    code_lit    CHAR(3),
    country_name CHAR(50)
  END RECORD,
  pers_info   ARRAY [5] OF RECORD
    f_name VARCHAR (20),
    sex VARCHAR (6),
    d_birth DATE,
    mar_status VARCHAR (7),
    country VARCHAR (15)
  END RECORD

MAIN

# This application uses widgets non-displayable
# in character mode
  IF fgl_fglgui()= 0 THEN
    CALL fgl_winmessage("Wrong mode",
      "Run this application in the GUI mode.", "info")
    EXIT PROGRAM
  END IF

OPTIONS
# We assign the keys instead of the default ones for the user convenience
  ACCEPT KEY F3,
  INSERT KEY F2,
  DELETE KEY F8

MENU "
  COMMAND "Editing Array"
    "The INPUT ARRAY execution with a partially initialized program array"
    CALL edit_arr()

  COMMAND "Clean Array"
    "The INPUT ARRAY with uninitialized program array"
    CALL clean_arr()

  COMMAND "Widget Array"
    "The INPUT ARRAY with graphical form widgets"
    CALL widget_arr()
```



```
COMMAND "Exit" "Exit the program"
        EXIT PROGRAM

END MENU

END MAIN

#####
# This function allows the user to perform input into a program array which
# has some values. It also applies some restrictions as to the inputted values.
#####
FUNCTION edit_arr()
    DEFINE
        i,
        ret_code,           -- the variable for the code returned by a function
        is_delete,          -- the row delete indicator
        is_change,          -- the row modification indicator
        dim_of_arr,         -- the variable to be used as the size of the array
        num_arr_row,        -- the variable to contain the number of the row of
                            -- the program array where the cursor is located
        cnt_arr_elm INTEGER, -- the variable to contain the number of elements
                            -- contained in the program array
        answer   CHAR(3),   -- This variable will be used to process the user's
                            -- responses
        err_mess  CHAR(80)  -- the text of the error message

OPEN WINDOW w_countr_lst_1 AT 2,2 WITH FORM "standard_array"
    ATTRIBUTE (FORM LINE FIRST, COMMENT LINE FIRST+1) -- this form contains
                                                    -- screen array
                                                    -- s_country_list

DISPLAY "!" TO info_but
DISPLAY "!" TO insert_but
DISPLAY "!" TO save_but
DISPLAY "!" TO del_but
DISPLAY "!" TO quit_but

# We assign the size of array dim_of_arr to variable a_country_list to be
# able to refer to the array elements dynamically.
LET dim_of_arr = 15

# We assign values to the first 10 elements of the array,
# whereas the total size of the array is 15.
LET a_country_list[1].country_id = "004"
LET a_country_list[1].code_lit = "AFG"
LET a_country_list[1].country_name = "Afghanistan"
LET a_country_list[2].country_id = "008"
LET a_country_list[2].code_lit = "ALB"
LET a_country_list[2].country_name = "Albania"
LET a_country_list[3].country_id = "010"
LET a_country_list[3].code_lit = "ATA"
LET a_country_list[3].country_name = "Antarctica"
LET a_country_list[4].country_id = "012"
LET a_country_list[4].code_lit = "DZA"
```



```

LET a_country_list[4].country_name = "Algeria"
LET a_country_list[5].country_id = "016"
LET a_country_list[5].code_lit = "ASM"
LET a_country_list[5].country_name = "American Samoa"
LET a_country_list[6].country_id = "020"
LET a_country_list[6].code_lit = "AND"
LET a_country_list[6].country_name = "Andorra"
LET a_country_list[7].country_id = "024"
LET a_country_list[7].code_lit = "AGO"
LET a_country_list[7].country_name = "Angola"
LET a_country_list[8].country_id = "028"
LET a_country_list[8].code_lit = "ATG"
LET a_country_list[8].country_name = "Antigua And Barbuda"
LET a_country_list[9].country_id = "031"
LET a_country_list[9].code_lit = "AZE"
LET a_country_list[9].country_name = "Azerbaijan"
LET a_country_list[10].country_id = "032"
LET a_country_list[10].code_lit = "ARG"
LET a_country_list[10].country_name = "Argentina"

CALL set_count(10)           -- as we have 10 elements with assigned values, we
                            -- set 10 as the total array row count

# The values which are assigned to the program array elements will be
# displayed to the screen array during the input, as the WITHOUT DEFAULTS
# keywords are included.
INPUT ARRAY a_country_list WITHOUT DEFAULTS FROM s_country_list.*


BEFORE INPUT
    LET is_change = FALSE -- we set the change indicator to FALSE(0)
                            -- initially which means that no rows has yet
                            -- been changed

BEFORE ROW
    LET is_delete = FALSE -- we set the delete indicator to FALSE(0) also
                            -- to specify that no rows has yet been deleted

BEFORE INSERT
    CALL fgl_winmessage("BEFORE INSERT", "A row will be inserted", "info")

# Before a row is deleted we check whether there are still rows left in
# the program array. If there are rows and the deleted row is not empty
# 4GL generates the prompt message to confirm the deleting.
BEFORE DELETE -- executed before a row is actually deleted
    LET num_arr_row = arr_curr()
    IF arr_count() > 0 AND
        # We use the variable instead of the array size to specify the array
        # element depending on the value returned by the arr_curr() function
        # and make sure that the array record is not empty
        (length(a_country_list[num_arr_row].country_id) <> 0 OR
         length(a_country_list[num_arr_row].code_lit) <> 0 OR
         length(a_country_list[num_arr_row].country_name) <> 0) THEN

        CALL fgl_winquestion("DELETE?",
                            "Do you really want to delete this row?", "Yes", "Yes|No",
                            "question",1) RETURNING answer

```



```
IF answer MATCHES "Yes" THEN
    CONTINUE INPUT
ELSE
    CANCEL DELETE -- though the Delete key was pressed, nothing
                    -- will be deleted in this case
END IF
END IF

ON KEY(F1)
# When F1 or the Info button is pressed, the general information about the
# arrays is returned by some built-in functions from within function info()
CALL info()

ON KEY (F10)
# When this key or the Quit button is pressed, the program control is
passed to the
# AFTER INPUT clause.
GOTO lab_exit

AFTER FIELD country_id,code_lit,country_name
# After the cursor leaves one of the fields specified in this clause
# 4GL verifies whether the data were changed. If they were changed, we
# assign the value TRUE(1) to the change indicator variable is_change
IF FIELD_TOUCHED(country_id) OR
    FIELD_TOUCHED(code_lit) OR
    FIELD_TOUCHED(country_name)
THEN LET is_change = TRUE
END IF

# After a row has been changed, we also set the change indicator variable
# is_change to TRUE(1)
AFTER INSERT
LET is_change = TRUE

# After a row has been deleted, we set the delete indicator variable
# is_delete to TRUE(1)
AFTER DELETE
LET is_delete = TRUE
LET is_change = TRUE
# If all the rows have been deleted, we inform the user about it.
IF arr_count() = 0 THEN
    CALL fgl_message_box ("All rows have been removed")
END IF

AFTER ROW
# In this clause 4GL verifies the values entered in each
# field of the screen record row
LET num_arr_row = arr_curr()
LET cnt_arr_elm = arr_count()
IF length(a_country_list[num_arr_row].country_id)      = 0 AND
    length(a_country_list[num_arr_row].code_lit)        = 0 AND
    length(a_country_list[num_arr_row].country_name)     = 0
THEN          -- this clause is executed if the row is empty
              -- (no values have been entered)
```



```

IF num_arr_row >= arr_count()
THEN IF FGL_LASTKEY() = FGL_KEYVAL("Up") OR
    FGL_LASTKEY() = FGL_KEYVAL("Left")
    THEN -- this clause is executed if UP or LEFT key has been
    -- pressed - the input is continued
    CONTINUE INPUT
    ELSE -- Otherwise the data entry is blocked
        IF arr_count() <> 0 THEN
            CALL fgl_message_box ("You cannot input an empty row")
            NEXT FIELD country_id
        END IF
    END IF
    ELSE -- The data entry is blocked
        CALL fgl_message_box ("You cannot input an empty row")
        NEXT FIELD country_id
    END IF
ELSE IF is_delete = FALSE
    THEN IF is_change = TRUE
        THEN -- If the data in a row were modified, but the row
        -- wasn't deleted, we check the modified values using
        -- check_data_row function declared at the end of the
        -- module
        CALL check_data_row("country_id",
            a_country_list[num_arr_row].country_id,
            num_arr_row, cnt_arr_elm)
        RETURNING ret_code,err_mess
        IF ret_code < 0
            THEN -- It is executed if an invalid value was entered
            -- for the country id
            CALL fgl_message_box (err_mess)-- we display the
            -- error message
            -- returned by the
            -- function
            NEXT FIELD country_id
        END IF

        CALL check_data_row("code_lit",
            a_country_list[num_arr_row].code_lit,
            num_arr_row,cnt_arr_elm)
        RETURNING ret_code,err_mess
        IF ret_code < 0
            THEN -- It is executed if an invalid value was entered
            -- for the country code
            CALL fgl_message_box (err_mess)
            NEXT FIELD code_lit
        END IF
        CALL check_data_row("country_name",
            a_country_list[num_arr_row].country_name,
            num_arr_row, cnt_arr_elm)
        RETURNING ret_code,err_mess
        IF ret_code < 0
            THEN -- It is executed if an invalid value was entered
            -- for the country name
            CALL fgl_message_box (err_mess)
            NEXT FIELD country_name

```



```
        END IF
    END IF
END IF
END IF

AFTER INPUT
LABEL lab_exit:
IF is_change THEN -- is clause is executed if the data were changed
    IF FGL_LASTKEY() = FGL_KEYVAL("F10")THEN -- if the key which
        -- invoke the AFTER INPUT clause was F10
        CALL fgl_winquestion("DATA CHANGED", "The data was changed" ||
            "\nDo you really want to quit?", "Yes", "Yes|No", "question", 1)
        RETURNING answer
    IF answer MATCHES "Yes" THEN
        EXIT INPUT
    ELSE
        CONTINUE INPUT
    END IF
END IF

# If the key which invoked the AFTER INPUT clause was F3 or
# if the Save button was pressed
IF FGL_LASTKEY() = FGL_KEYVAL("F3") THEN
    CALL fgl_winmessage ("Saved", "The data was saved", "info")

    # We assign NULL to all the array elements
    # which were not assigned any values by the user
    FOR i = arr_count() + 1 TO dim_of_arr
        INITIALIZE a_country_list[i].* TO NULL
    END FOR

    # We display the program array to the screen array to show the
    # modified values
    CALL disp_a_country_list()
    OPTIONS ON CLOSE APPLICATION STOP
    LET is_change = FALSE -- we set the change indicator back
    CONTINUE INPUT -- to FALSE to restore status quo
END IF
-- and remain in the input after
-- saving the data

ELSE -- if the data were not changed
    IF FGL_LASTKEY() = FGL_KEYVAL("F10")THEN
        EXIT INPUT
    END IF
    IF FGL_LASTKEY() = FGL_KEYVAL("F3") THEN
        CONTINUE INPUT -- the input is resumed, because there
    END IF
        -- are no values to accept
END IF

END INPUT
CLOSE WINDOW w_countr_lst_1
CLEAR SCREEN
```



```
END FUNCTION

#####
# This function performs input into a non-initialized program array. This is a
# simpler version of input which does not contain as many restrictions
#####
FUNCTION clean_arr()
DEFINE
    i      INTEGER

    # We initialize all the elements of the program array to NULL
FOR i = 1 TO 10
    INITIALIZE a_country_list[i].* TO NULL
END FOR

OPEN WINDOW w_countr_lst_1 AT 2,2 WITH FORM "standard_array"
    ATTRIBUTE (FORM LINE FIRST,COMMENT LINE FIRST+15)
DISPLAY "!" TO info_but
DISPLAY "!" TO insert_but
DISPLAY "!" TO save_but
DISPLAY "!" TO del_but
DISPLAY "!" TO quit_but
    # The INPUT ARRAY without the WITHOUT DEFAULTS keywords
    # is typically used to input the data into an empty array
INPUT ARRAY a_country_list FROM s_country_list.*

BEFORE INPUT
# We assign values to the elements of the first program array row
    LET a_country_list[1].country_id    = "004"
    LET a_country_list[1].code_lit     = "AFG"
    LET a_country_list[1].country_name = "Afghanistan"
    CALL fgl_dialog_setcurrline(1,1) -- And display them to the first row of
                                    -- the screen array

ON KEY(F1)
    CALL info()

ON KEY (F10)
    EXIT INPUT

AFTER INPUT
    CALL fgl_wimmessage("Saved","The data was saved", "info")
    CALL disp_a_country_list()-- the entered values are displayed to the
                            -- screen array using this function,
                            -- and the input continues
    OPTIONS ON CLOSE APPLICATION STOP
    CONTINUE INPUT
END INPUT

CLOSE WINDOW w_countr_lst_1

CLEAR SCREEN
END FUNCTION

#####
```



```
# This function performs input from a screen array consisting of different
# types of widgets.
#####
FUNCTION widget_arr()

# We open a window with the second form
OPEN WINDOW form_window AT 2,2 WITH FORM "widget_array" ATTRIBUTE (FORM LINE
FIRST)
    DISPLAY "!" TO save_but
    DISPLAY "!" TO quit_but
    DISPLAY "!" TO del_but
    DISPLAY "!" TO insert_but

# and activate it for input. This time the screen array includes various
# graphical form widgets
INPUT ARRAY pers_info FROM s_pers_info.*

ON KEY (F10)
    EXIT INPUT

AFTER INPUT

OPTIONS
    ON CLOSE APPLICATION KEY F3 -- prevent the Close(x) button from
                                -- terminating the application

OPEN WINDOW win_info AT 6,12 WITH 15 ROWS , 60 COLUMNS
    ATTRIBUTE (BORDER)
DISPLAY "Records of the program array pers_info[]:""
    AT 2,14 ATTRIBUTE (BOLD, BLUE, UNDERLINE)
DISPLAY "pers_info[1]:" , " ", pers_info[1].f_name, " ",
        pers_info[1].sex, " ", pers_info[1].d_birth, " ",
        pers_info[1].mar_status, " ", pers_info[1].country AT 4,2
DISPLAY "pers_info[2]:" , " ", pers_info[2].f_name, " ",
        pers_info[2].sex, " ", pers_info[2].d_birth, " ",
        pers_info[2].mar_status, " ", pers_info[2].country AT 5,2
DISPLAY "pers_info[3]:" , " ", pers_info[3].f_name, " ",
        pers_info[3].sex, " ", pers_info[3].d_birth, " ",
        pers_info[3].mar_status, " ", pers_info[3].country AT 6,2
DISPLAY "pers_info[4]:" , " ", pers_info[4].f_name, " ",
        pers_info[4].sex, " ", pers_info[4].d_birth, " ",
        pers_info[4].mar_status, " ", pers_info[4].country AT 7,2
DISPLAY "pers_info[5]:" , " ", pers_info[5].f_name, " ",
        pers_info[5].sex, " ", pers_info[5].d_birth, " ",
        pers_info[5].mar_status, " ", pers_info[5].country AT 8,2
DISPLAY "Press any key to return to INPUT" AT 11,15
    ATTRIBUTE (BOLD, BLUE, UNDERLINE)

CALL fgl_getkey()
CLOSE WINDOW win_info
OPTIONS
    ON CLOSE APPLICATION STOP -- reset the Close(x) button to normal
                                -- functioning
CONTINUE INPUT
```



```
END INPUT
CLOSE WINDOW form_window

END FUNCTION
#####
# This function is used to display the elements of the first program array.
# It is invoked from the MAIN block whenever we need to display changed values.
#####
FUNCTION disp_a_country_list()
DEFINE
    num_key  INTEGER

    # This ensures that pressing the Close(x) button on the window
    # will not terminate the application execution
    # it is reset to STOP in the calling routine
OPTIONS
    ON CLOSE APPLICATION KEY F3

OPEN WINDOW w_mess AT 3,1 WITH 21 ROWS, 80 COLUMNS ATTRIBUTE (BORDER)
DISPLAY "Records of the program array a_country_list[]" AT 2,17
ATTRIBUTE(BLUE, BOLD, UNDERLINE)
DISPLAY "a_country_list[1] = ", a_country_list[1].country_id, " ",
a_country_list[1].code_lit, " ", a_country_list[1].country_name AT 4,2
    DISPLAY "a_country_list[2] = ", a_country_list[2].country_id, " ",
a_country_list[2].code_lit, " ", a_country_list[2].country_name AT 5,2
    DISPLAY "a_country_list[3] = ", a_country_list[3].country_id, " ",
a_country_list[3].code_lit, " ", a_country_list[3].country_name AT 6,2
    DISPLAY "a_country_list[4] = ", a_country_list[4].country_id, " ",
a_country_list[4].code_lit, " ", a_country_list[4].country_name AT 7,2
    DISPLAY "a_country_list[5] = ", a_country_list[5].country_id, " ",
a_country_list[5].code_lit, " ", a_country_list[5].country_name AT 8,2
    DISPLAY "a_country_list[6] = ", a_country_list[6].country_id, " ",
a_country_list[6].code_lit, " ", a_country_list[6].country_name AT 9,2
    DISPLAY "a_country_list[7] = ", a_country_list[7].country_id, " ",
a_country_list[7].code_lit, " ", a_country_list[7].country_name AT 10,2
    DISPLAY "a_country_list[8] = ", a_country_list[8].country_id, " ",
a_country_list[8].code_lit, " ", a_country_list[8].country_name AT 11,2
    DISPLAY "a_country_list[9] = ", a_country_list[9].country_id, " ",
a_country_list[9].code_lit, " ", a_country_list[9].country_name AT 12,2
    DISPLAY "a_country_list[10] = ", a_country_list[10].country_id, " ",
a_country_list[10].code_lit, " ", a_country_list[10].country_name AT 13,2
    DISPLAY "a_country_list[11] = ", a_country_list[11].country_id, " ",
a_country_list[11].code_lit, " ", a_country_list[11].country_name AT 14,2
    DISPLAY "a_country_list[12] = ", a_country_list[12].country_id, " ",
a_country_list[12].code_lit, " ", a_country_list[12].country_name AT 15,2
    DISPLAY "a_country_list[13] = ", a_country_list[13].country_id, " ",
a_country_list[13].code_lit, " ", a_country_list[13].country_name AT 16,2
    DISPLAY "a_country_list[14] = ", a_country_list[14].country_id, " ",
a_country_list[14].code_lit, " ", a_country_list[14].country_name AT 17,2
    DISPLAY "a_country_list[15] = ", a_country_list[15].country_id, " ",
a_country_list[15].code_lit, " ", a_country_list[15].country_name AT 18,2
DISPLAY "Press any key to return to INPUT" AT 20,24
ATTRIBUTE(BLUE, BOLD, UNDERLINE)
LET num_key = fgl_getkey()
```



```
CLOSE WINDOW w_mess

END FUNCTION

#####
# This function is used to verify the changed or newly entered values
#####
FUNCTION check_data_row(name_field, value_field, num_arr_row, cnt_arr_elm)
DEFINE
    name_field,                      -- the name of the field will be assigned to this
    -- variable to make the function more flexible
    value_field CHAR(252),           -- the variable to store the value entered
    num_arr_row,                      -- the variable to store the number of the row
    -- where the verified value is located
    cnt_arr_elm,                     -- the variable to store the number of elements
    -- contained in the program array
    i          INTEGER,
    ret_code   INTEGER,
    err_mess   CHAR(80)

# We initialize the variables which concern the possible errors to NULL to
# prevent false values
LET ret_code = 0
LET err_mess = NULL

CASE          -- we use the CASE statement to specify the actions for each
              -- field separately
WHEN name_field = "country_id"

    # The country code must contain at least 3 digits, if it is shorter,
    # the value is invalid
    IF length(value_field) < 3
    THEN -- if the value is invalid we assign corresponding values to the
          -- returned code and to the error message
          -- these value will be returned to the MAIN routine
        LET ret_code = -1
        LET err_mess = "The country code must contain at least 3 digits"
        RETURN ret_code,err_mess
    END IF

    # We check whether another array element has the same value
    # (as we need only unique values)
    FOR i = 1 TO cnt_arr_elm

        # We skip the row of the array where the value
        # under investigation is located to make the check valid
        IF i = num_arr_row THEN CONTINUE FOR END IF
        IF value_field = a_country_list[i].country_id
        THEN LET ret_code = -2
            LET err_mess = "Such country code already exists"
            RETURN ret_code,err_mess
        END IF
    END FOR
WHEN name_field = "code_lit" -- similar check is performed for code_lit
                           -- field
```



```

        IF length(value_field) < 3
    THEN LET err_mess = "Letter code must contain at least 3 characters "
        LET ret_code = -1
        RETURN ret_code,err_mess
    END IF
    FOR i = 1 TO cnt_arr_elm
        IF i = num_arr_row THEN CONTINUE FOR END IF
        IF value_field = a_country_list[i].code_lit
        THEN LET ret_code = -2
    LET err_mess = "Such letter code has already been introduced"
        RETURN ret_code,err_mess
    END IF
    END FOR
    WHEN name_field = "country_name" -- similar check is performed for
        -- country_name field

        # This field should contain a value, but its length is not preset
    IF length(value_field) = 0
    THEN LET err_mess = " Enter a value in the field "
        LET ret_code = -1
        RETURN ret_code,err_mess
    END IF
    FOR i = 1 TO cnt_arr_elm
        IF i = num_arr_row THEN CONTINUE FOR END IF
        IF value_field = a_country_list[i].country_name
        THEN LET ret_code = -2
            LET err_mess = " Such name of the country already exists "
            RETURN ret_code,err_mess
        END IF
    END FOR
    END CASE
    RETURN ret_code,err_mess -- We return the code and error message to
        -- the MAIN routine.
END FUNCTION

#####
# This function displays information about the program and screen arrays
#####
FUNCTION info()
    DEFINE
        num_arr_row,
        num_scr_row,
        cnt_arr_elm,
        cnt_scr_elm      INT

    LET num_arr_row = arr_curr()-- The current program array row
    LET num_scr_row = scr_line()-- The current screen array row
    LET cnt_arr_elm = arr_count()-- The total number of program array records
    LET cnt_scr_elm = fgl_scr_size("s_country_list")-- The total number of screen
        -- array lines
    DISPLAY BY NAME num_arr_row,num_scr_row,cnt_arr_elm,cnt_scr_elm
    SLEEP 3
    CLEAR num_arr_row
    CLEAR num_scr_row

```



```
CLEAR cnt_arr_elm
CLEAR cnt_scr_elm

END FUNCTION
```

Form Files

The first form "standard_array.per". The screen array consists of ordinary text fields.

```
DATABASE formonly
```

```
SCREEN
```

```
{
  \gp-----[f16 ]-----q\g
  \g|[f17][f18][f19 ]-----| \g
  \g|[f00][f01][f02 ]-----| \g
  \g|[f03 [f07 ]-----| \g
  \g|[f04 [f08 ]-----| \g
  \g|[f05 [f09 ]-----| \g
  \g|[f06 [f10 ]-----| \g
  \g|[f11 ][f12 ][f13 ][f14 ][f15 ]-----| \g
  \gb-----d\g
}
```

```
ATTRIBUTES
```

```
f00 = formonly.country_id TYPE CHAR, PICTURE ="###", UPSHIFT,AUTONEXT;
f01 = formonly.code_lit TYPE CHAR, PICTURE="AAA", UPSHIFT,AUTONEXT;
f02 = formonly.country_name TYPE CHAR, AUTONEXT;
f03 = formonly.prog_cur_row_lab, widget = "label",
      config = "The current row of the program array is:", COLOR = BLUE BOLD;
f04 = formonly.scr_cur_row_lab,widget = "label",
      config = "The current row of the screen array is:", COLOR = BLUE BOLD ;
f05 = formonly.prog_rec_num_lab,widget = "label",
      config = "The total number of program array records is:",
      COLOR = BLUE BOLD;
f06 = formonly.scr_rec_num_lab, widget = "label",
      config = "The total number of screen array records is:",
      COLOR = BLUE BOLD;
f07 = formonly.num_arr_row, CENTER, COLOR = BOLD GREEN;
f08 = formonly.num_scr_row, CENTER, COLOR = BOLD GREEN;
```



```
f09 = formonly.cnt_arr_elm, CENTER, COLOR = BOLD GREEN;
f10 = formonly.cnt_scr_elm, CENTER, COLOR = BOLD GREEN;
f11 = formonly.info_but, widget = "button", config="F1 {Info}", CENTER,
      COLOR = UNDERLINE BOLD;
f12 = formonly.insert_but, widget="button", config = "F2 {Insert}", CENTER,
      COLOR = UNDERLINE BOLD;
f13 = formonly.save_but, widget = "button", config = "F3 {Save}", CENTER,
      COLOR = UNDERLINE BOLD;
f14 = formonly.del_but, widget = "button", config = "F8 {Delete}", CENTER,
      COLOR = UNDERLINE BOLD;
f15 = formonly.quit_but, widget = "button", config = "F10 {Quit}", CENTER,
      COLOR = UNDERLINE BOLD;
f16 = formonly.c_list_lab, widget = "label", config = "COUNTRY LIST", CENTER,
      COLOR = CYAN BOLD REVERSE;
f17 = formonly.id_lab, widget = "label", config = "Id", CENTER,
      COLOR = BLUE BOLD;
f18 = formonly.lit_lab, widget = "label", config = "Lit", CENTER,
      COLOR = BLUE BOLD ;
f19 = formonly.c_name_lab, widget = "label", config = "Country Name",
      COLOR = BLUE BOLD;
```

INSTRUCTIONS

```
DELIMITERS " [ ] "

SCREEN RECORD s_country_list[10] (country_id THRU country_name)
```

This is "widget_array.per". The screen array is composed of various graphical form widgets: checkboxes, radio buttons, combo boxes, calendar widgets.

DATABASE formonly

SCREEN



```
\g| [f11 ][f12 ][f13 ][f14 ] |\g
\gb-----d\g
}
```

ATTRIBUTES

```
LAB = formonly.pers_inf_lab, widget = "label", config = "PERSONAL INFORMATION",
      CENTER, COLOR = CYAN BOLD REVERSE;
f01 = formonly.f_name_lab, widget="label", config = "NAME", CENTER,
      COLOR = BLUE BOLD;
f02 = formonly.sex_lab, widget = "label", config = "SEX", CENTER,
      COLOR = BLUE BOLD;
f03 = formonly.d_birth_lab, widget = "label", config = "DATE OF BIRTH", CENTER,
      COLOR = BLUE BOLD;
f04 = formonly.m_status_lab, widget = "label", config = "MARRIED", CENTER,
      COLOR = BLUE BOLD;
f05 = formonly.country_lab, widget = "label", config = "COUNTRY", CENTER,
      COLOR = BLUE BOLD;
f06 = formonly.f_name;
f07 = formonly.sex, widget="radio", config = "Male {Male} Female {Female}";
f08 = formonly.d_birth, widget="calendar";
f09 = formonly.mar_status, widget="check", config = "Married Single { }";
f10 = formonly.country, widget="combo",
      include = ("Argentina", "Bulgaria", "Canada", "France", "Germany"),
      class = "combo";
f11 = formonly.save_but, widget = "button", config = "F3 {Save}", CENTER,
      COLOR = UNDERLINE BOLD;
f12 = formonly.insert_but, widget = "button", config = "F2 {Insert}", CENTER,
      COLOR = UNDERLINE BOLD;
f13 = formonly.del_but, widget = "button", config = "F8 {Delete}", CENTER,
      COLOR = UNDERLINE BOLD;
f14 = formonly.quit_but, widget = "button", config = "F10 {Quit}", CENTER,
      COLOR = UNDERLINE BOLD;
```

INSTRUCTIONS

DELIMITERS "[]"

```
SCREEN RECORD s_pers_info [5] (f_name THRU country)
```



Using a Toolbar

When an application is run, Phoenix and Chimera provide the user with a default toolbar containing three buttons which are Accept, Cancel, and Help. The toolbar appears at the top of the window, above the menu line. It looks as follows:



You do not have to specify any ON KEY directives for the default toolbar buttons to make them influence the program workflow; 4GL reacts on them automatically. However, you can use ON KEY clauses to change the default behavior of these buttons.

Graphic clients also support a customizable toolbar which have user-defined buttons. You can create, alter, and remove this toolbar during the program execution; you can also dynamically change the set of buttons placed to the dynamic toolbar and the buttons functionality at any stage of the program execution.

To specify a dynamic toolbar you have to define its buttons within a form file. The toolbar is not visible and has no effect, if the application is run in the character mode.

Creating Dynamic Toolbars

A form file can have two sections that haven't been discussed before. They are KEYS and ACTIONS sections. These sections contain the default properties of the buttons that comprise the dynamic toolbar. The properties include the labels and icons the buttons are to have, and the order in which these buttons will be placed on the toolbar.

The buttons specified in the KEYS section must be associated with ON KEY clauses of those statements which can have these clauses. The buttons specified in the ACTIONS section, in their turn, must be associated with ON ACTION clauses. A button which is not associated with an ON KEY or ON ACTION clause, even though it may be specified in the form, will not be added to the toolbar.

The general syntax of a dynamic toolbar button specification is practically the same for both KEYS and ACTIONS sections. Whether you use the KEYS or the ACTIONS section depends only on whether you want to use the actions of the keys for button activation, but remember, that the ACTIONS section must follow the KEYS section, if your form file includes both:

```
KEYS
Key = "Button label [("icon")] [ORDER integer]
```

or

```
ACTIONS
```



Action = "Button label" [("icon")] [ORDER integer]

	<p>Note: The key or action specification must not end with a semicolon, even if there is more than one such specification.</p>
---	---

In the given scheme, *Key* or *Action* stands for the name of a key or an action which will be later associated with some program actions. Either should not be enclosed in quotation marks.

The Button Label

Button label stands for a character string which will be displayed to the button. The button label is also displayed as a tooltip when the mouse cursor is positioned over the button. If you don't want the button to have a label, you can include an empty string instead of the label. However, this will lead to the empty tooltip message.

You can switch off the tooltips by adding the following script option to a script file:

```
? .dynamicToolbar.show tooltips: false
```

You can also hide the button labels using the following script option:

```
? .dynamicToolbar.hidelabels: true
```

These script options are described in more details later in this chapter.

The Button Icon

The *icon* stands for the name of a file containing a picture you want to become a button icon, preceded with its path, if necessary. The default size of the button icon is 16x16. If you use larger icons, they can be truncated or distorted in other ways. If you use smaller pictures, they will be displayed to the top left corners of the places allocated for the icons.

However, there is a way to change the size of the displayed button icons. This can be done by adding two script options to the script file. One of these options changes the width of the icon, and the other changes its height. These script options are to be specified as follows:

```
Program.dynamicToolbar.iconHeight: size
```

```
Program.dynamicToolbar.iconWidth: size
```

Where *size* stands for an integer indicating the new value of the icon height or width parameter. It is advisable that you specify icons with equal height and width and of standard sizes like 9x9, 16x16, 32x32. You can make icons larger and with different height and width sizes, but it can lead to unexpected display results. When you specify a new icon size, you should specify the icon images that fit this size.



The ORDER Keyword

The ORDER keyword and an integer value following it are used to specify the order in which the buttons will be placed on the toolbar. The program places the buttons on the toolbar one by one from left to right according to the order values of the buttons (in the ascendant order). If one or several buttons have no ORDER parameter, they will appear at the end of the toolbar in the order in which they are specified in the .per file. When you specify dynamic toolbar buttons in a .per file, you should keep to the following rule: the KEYS section should be specified after the INSTRUCTIONS section, and the ACTIONS section should follow the KEYS section. This means that if the ORDER keywords are absent, the buttons with keys specified in the KEYS clause will be displayed at the beginning of the toolbar in the order of their appearance in the KEYS section. The buttons with actions will be placed after them.

As it can be seen from the information above, you have to make two steps to create a toolbar button:

- Specify a key or an action name associated with the button, the button name, icon, and order
- Bind the button to an ON KEY or an ON ACTION clause of an input or output statement in one of the program modules.

	Note: Remember, that the toolbar specified in a form file will be displayed only when the corresponding form is opened and displayed to the screen.
--	--

Now we will illustrate these instructions with a simple example of a toolbar creation. Let's create a form file named "Toolbar_form". This Form file will have two simple form fields and a frame:

```
DATABASE formonly

SCREEN {

    \gp-----q\g
    \|                                | \g
    \|                                | \g
    \| [f001      ]      [f002      ] | \g
    \|                                | \g
    \|-----d\g
}

ATTRIBUTES
    f001 = formonly.f001;
    f002 = formonly.f002;
```



KEYS

```
F1 = "F1 button"("f1_button.png")ORDER 1
```

ACTIONS

```
act = " " ("act_button.png")ORDER 2
act2 = "ACT action" ORDER 3
```

In this example, we will create three toolbar buttons - one of them will be associated with a key and the others - with actions. When you define a toolbar button, you don't have to add a special form widget for it either to the SCREEN section or to the ATTRIBUTES section. The program automatically adds the button to the toolbar.

- The button associated with the key F1 has a label and an icon and will be the first in the buttons list when the toolbar is displayed.
- The button associated with the action *act* has an empty label, but it also has an icon and will be displayed after the F1 button.
- The button associated with the action *act2* doesn't have any icon, but it does have a label.



Note: Before you specify a picture which is to become a button icon, don't forget to import the picture to your project and to add it to the program requirements by the means of the *Add requirement* option.

Now, when all the buttons are specified, it's time to activate them in the program:

MAIN

```
DEFINE a,b CHAR
```

```
OPEN WINDOW w1 AT 2,2 WITH FORM "my_form"
```

```
ATTRIBUTES (BORDER)
```

```
OPTIONS PROMPT LINE LAST -1
```

```
INPUT a,b FROM f001, f002
```

```
ON KEY (F1)
```

```
    CALL fgl_winmessage ("Toolbar option",
                          "F1 button is pressed", "Information")
```

```
ON ACTION (act)
```

```
    CALL fgl_winmessage ("Toolbar option ",
                          "ACT action is performed", "Information")
```

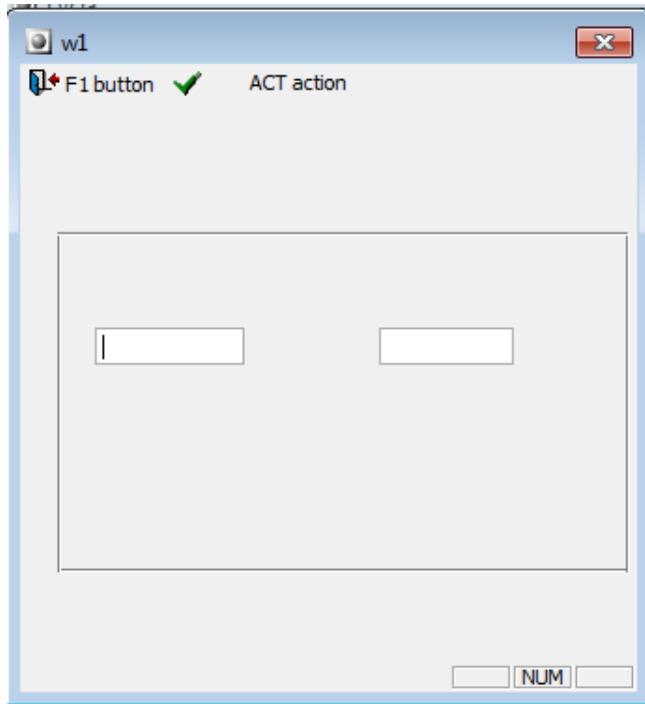
```
ON ACTION (act2)
```

```
    CALL fgl_winmessage ("Toolbar option ",
```



```
"ACT2 action is performed", "Information")  
END INPUT  
END MAIN
```

When you run the program, you will see the following picture:



The click on a toolbar button will result in the corresponding ON KEY or ON ACTION clause activation. If a toolbar button is associated with a key, the actions specified for this button can be invoked either by a keypress on the keyboard (in our case by pressing F1) or by clicking the corresponding toolbar button.

Field Specific Toolbar Buttons

You can also specify some toolbar buttons that will appear only when the cursor is positioned to a particular form field. To do this, you have to specify a KEY or ACTION clause as a part of the field attributes. Then the key specification should be included into the form field declaration in the ATTRIBUTES section and have the following format:

```
KEY | ACTION key_name | action_name =  
"Button label" [("icon")] [ORDER integer]
```

In this case the key or action specification are just parts of the field declaration and should be separated by commas from other clauses of the same declaration.



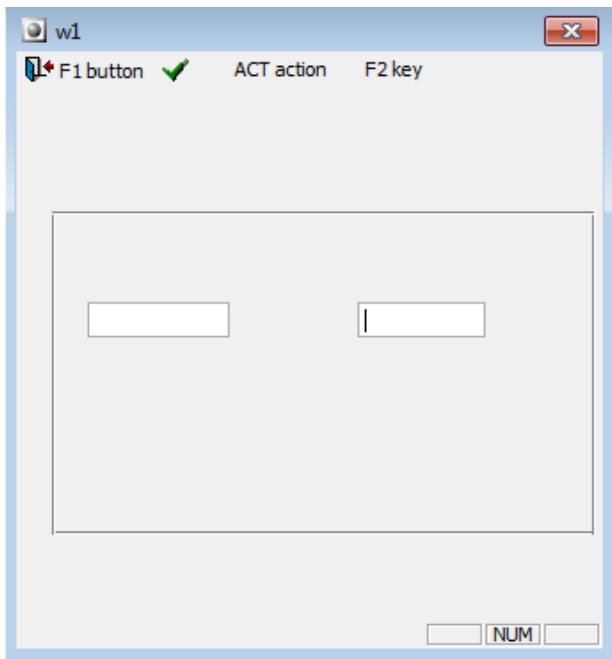
For example:

```
ATTRIBUTES
f002 = formonly.f002, KEY F2 = "F2 key" ORDER 4;
```

Don't forget to specify the actions for the button associated with the key F2:

```
...
ON KEY (F2) CALL fgl_winmessage ("Caused by a titlebar",
    "F2 key button is pressed", "Information")
```

Now, when the cursor moves to the field f002 during the input, the fourth button will appear on the toolbar. This button will be removed as soon as the cursor leaves the field:



Modifying the Dynamic Toolbar at Runtime

The toolbar specified in the form file is named a *dynamic* toolbar due to the possibility to change the toolbar view and options at runtime. We have already discussed how buttons can be added depending on the context during the program execution. However, there are several more ways to modify the dynamic toolbar.

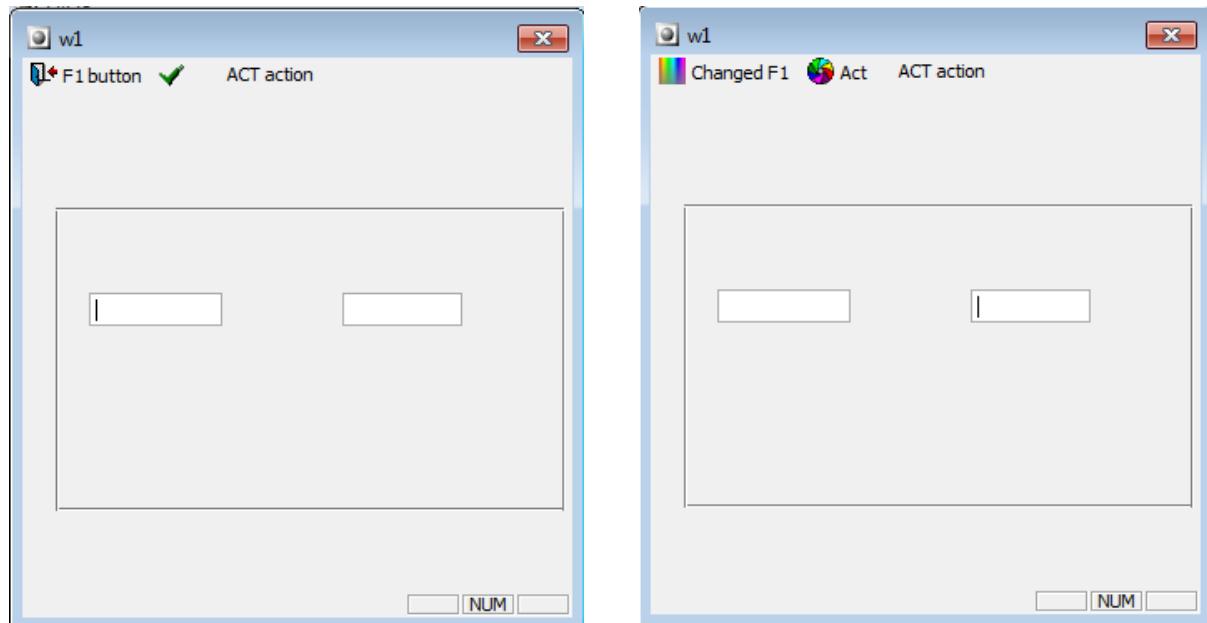


Changing Button Labels

You can use the KEY field attribute not only to add a button when the cursor moves to the corresponding field, but also to change the parameters of the existing buttons. To do this, you have to specify the key or action name of the existing button and specify its new label, image, etc:

```
ATTRIBUTES
f002 = foronly.f002,
KEY F1 = "Changed F1" ("ico16/gd_ic_color1.ico") ORDER 1,
ACTION act = "Act" ("ico16/gd_ic_color2.ico") ORDER 2;
```

If the f002 field has attributes as it is shown above, the toolbar view will change when the cursor moves to the field f002:



You should keep in mind that if you specify new properties for an existing button, *all* its default properties are ignored. It means that, for example, if a button has an icon by default, and you do not specify any icon in the KEY or ACTION form field attribute, the button will be displayed without the icon when the user positions the cursor to the corresponding field.

You can also use 4GL functions to add, delete, or change button labels. These functions can be invoked within the user interaction statements, such as INPUT, INPUT ARRAY, etc. as well as in parts of the source code which do not imply dialog with the user.



The functions which can be used to modify the dynamic toolbar have two variants of reference:

- *Fgl_functionName()*
- *Fgl_dialog_functionName()*

The first variant can be used to influence the toolbar from any part of the source code. If the *dialog* keyword is added to the function name, such function affects only the toolbar in the currently active form.

If you want to return the text that the button label currently contains, use the *fgl_getkeylabel("keyname")* or *fgl_dialog_getkeylabel("keyname")* function. The function returns a character string containing the label of the specified toolbar button. For example, if a button is specified as:

```
ATTRIBUTES
f001 = formonly.f001,
key F1 = "My key" ORDER 4;
```

the line

```
LET key_name = fgl_dialog_getkeylabel("F1")
```

will assign the value "My key" to the character variable *key_name*. To find the text of a label assigned to an action button, use the *fgl_getactionlabel("action")* or *fgl_dialog_getactionlabel("action")* function.

If you want to set a new text label for a button which does not have any icon, use the *fgl_setkeylabel(keyname, "label")* or *fgl_dialog_setkeylabel(keyname, "label")* and *fgl_setactionlabel(keyname, "action")* or *fgl_dialog_setactionlabel(keyname, "action")* functions.

```
CALL fgl_dialog_setkeylabel("F1", "Modified F1")
CALL fgl_dialog_setactionlabel("act", "Modified act")
```

The *fgl_[dialog_]setkeylabel()* and *fgl_[dialog_]setactionlabel()* functions can also take optional arguments - the *icon* and the *order*. The *icon* parameter follows the argument containing the new label and is to be represented by a quoted character string or a variable which contains such string. You have to add all the icons that can be specified in the *icon* parameter to the program and compile it. If the icon file specified in the *icon* argument is not attached to the program, it won't be displayed to the label.

The *order* argument should contain an integer value or a variable containing such value which indicates where on the toolbar the option will be moved.

```
CALL fgl_dialog_setactionlabel("nav_first", "New Label", icon_path, 40)
```



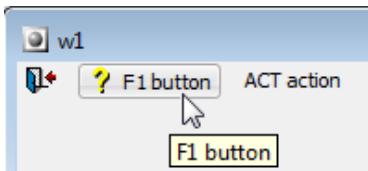
Both *icon* and *order* arguments are optional. However, if the toolbar button doesn't have any icon and you want to change the button position, you should pass the *icon* argument as an empty string.

```
CALL fgl_dialog_setactionlabel("nav_first", "New Label", "", 40)
```

You should also remember that you have to specify both icon and order arguments if the button to be changed has an icon and a specified position . If you don't do that, the icon will be removed from the button and the button itself will be moved to the left of the toolbar.

Hiding Toolbar Labels and Tooltips

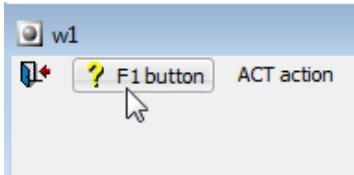
The button label is usually displayed as a tooltip when the mouse cursor is positioned over the button. In fact, there is no practical use of having one and the same information displayed twice:



GUI clients support script options that hide either button label or tooltip, depending on your objectives. To hide a tooltip appearing under the corresponding button, use the *dynamictoolbar.showtooltips: false* option. In our example, this option can be specified as:

```
Myprog.dynamictoolbar.showtooltips: false
```

This option can be applied only to all the toolbar buttons together, you cannot turn on the tooltips for some buttons and turn them off for the others. After this script option is added to the script file, no tooltips will appear:

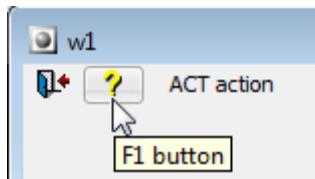


It is also possible to hide button labels. This can be done for aesthetic or functional purposes (for example, if you want more buttons to be displayed to the toolbar at once).

To hide the button labels, you should use the *dynamictoolbar.hidelabels: false* option:

```
Lycia.dynamictoolbar.hidelabels: true
```

When you apply this option, you will get the toolbar without labels. In this case it is advisable not to switch the tooltips off. If a toolbar button has no icon but has a label, the label will not be removed even if the *hidelabels* option is set to *false*. When the application is run, the toolbar will look as follows:

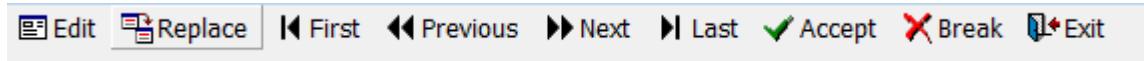


Both *hidelabels* and *show tooltips* options have effect only on the application level and cannot vary depending on the context. Their effect is permanent and cannot be cancelled at runtime.

Adding Toolbar Dividers

Different toolbar buttons are usually grouped by their functions. The buttons of one group are typically placed near each other on the toolbar. Querix 4GL supports functions which add a graphical divider between toolbar buttons or group of toolbar buttons. These are the *fgl_keydivider()* and *fgl_dialog_keydivider()* functions.

You have already seen a toolbar without any dividers. Here is one more example, a toolbar with nine buttons and no dividers:



It would be convenient to add some dividers to this toolbar. To do it, you should know the order number specified for each button. In our case, the buttons are defined as follows:

```
ACTIONS
record_edit = "Edit"("icos/record_edit01.ico")ORDER 101
replace = "Replace"("icos/replace01.ico")ORDER 102
nav_first = "First"("icos/nav_first01.ico")ORDER 105
nav_prev = "Previous"("icos/nav_prev_fast01.ico")ORDER 106
nav_next = "Next"("icos/nav_next_fast01.ico")ORDER 107
nav_last = "Last"("icos/nav_last01.ico")ORDER 108
accept = "Accept"("icos/accept01.ico")ORDER 110
break = "Break"("icos/break01.ico")ORDER 112
exit = "Exit"("icos/exit01.ico")ORDER 113
```

The syntax of the *keydivider* functions invocation is as follows:

```
CALL fgl_[dialog_]keydivider(order)
```

The *order* here stays for an integer value (or a variable containing such value) which is more than the ORDER number of the *left* button and less than that of the *right* button. When saying the *left* and the *right* buttons we mean the buttons between which the divider is to be placed. Thus you can see that in order to be



able to place the dividers, the order numbers of the buttons must not be strictly consecutive - there must be some space left for the dividers. E.g. button "Last" with ORDER 108 and button "Accept" with ORDER 110 allow to place the divider at 109 position.

If we want to place a divider between the *Replace* (ORDER 102) and the *First* (ORDER 105) buttons, we have to pass an argument which is more than 102 and less than 105. If the argument passed to the *fgl_[dialog_]keydivider()* function matches one of the numbers used in an ORDER clause, the delimiter will not be displayed.

Therefore, the following code will place dividers between the *Replace* and the *First* and between the *Last* and the *Accept* buttons:

```
DEFINE a,b,c VarCHAR(20),
      divid INT
LET divid = 104

CALL fgl_keydivider(divid) -- places the divider between the Replace and
the
                           -- First buttons

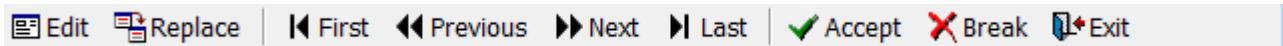
CALL fgl_dialog_keydivider(110) -- places no divider between the Accept
and
                           -- Break buttons, because the dialog
prefix is
                           -- used

OPEN WINDOW w1 AT 3,3 WITH FORM "my_form" ATTRIBUTES (BORDER)

CALL fgl_dialog_keydivider(109) -- places the dividers between the Last
and the
                           -- Accept keys

INPUT...
END INPUT
```

When you run the program, you will see the toolbar as follows:



This toolbar has two dividers.



Example

This application illustrates the toolbars application and modifying at runtime.

```
#####
# This is the demo program, illustrating how a dynamic toolbar can be created
# and modified by different statements and script options
#####

MAIN
DEFINE val1, val2, val3,
      val4, val5          CHAR(15),
      answer               INTEGER

OPTIONS INPUT WRAP

OPEN WINDOW win1 AT 1,1 WITH FORM "toolbar_form"

CALL fgl_keydivider(5) -- create dividers
CALL fgl_keydivider(45) -- on the form toolbar

# Here, the input begins and the toolbar buttons
# become active

INPUT val1, val2, val3, val4 FROM f001, f002, f003, f004
BEFORE INPUT

DISPLAY "!" TO b001
DISPLAY "!" TO b002
# Most of the ON KEY clauses reference toolbar buttons
# located on the screen form with the help of the
# KEYS and ACTIONS sections

# The CONTROL-R key (Restart option) is used to start the
# input from the very beginning with all values inputted before
# being initialized to null
ON KEY (CONTROL-R)
  INITIALIZE val1 TO NULL
  INITIALIZE val2 TO NULL
  INITIALIZE val3 TO NULL
  CLEAR FORM
  DISPLAY "!" TO b001
  DISPLAY "!" TO b002
  NEXT FIELD f001
  CONTINUE INPUT

# The CONTROL-E key (Exit option) allows the user to exit the program
ON KEY (CONTROL-E)
  LET answer = fgl_message_box("Exit",
                               "Do you want to exit the demo?", 4)
  IF answer = 1 THEN
    EXIT PROGRAM
  ELSE CONTINUE INPUT
END IF
```



```
BEFORE FIELD f001, f002
  DISPLAY "!" TO b001
  DISPLAY "!" TO b002

BEFORE FIELD f003, f004
  DISPLAY "*" TO b001
  DISPLAY "*" TO b002

ON ACTION (nav_first) NEXT FIELD f001 -- the 'First' toolbar option moves the
                                         -- cursor to the first field
ON ACTION (nav_last) NEXT FIELD f004 -- the 'Last' toolbar option moves the
                                         -- cursor to the last field
ON ACTION (nav_prev) NEXT FIELD PREVIOUS -- the 'Previous' toolbar option
                                         -- moves the cursor to the previous
                                         -- field
ON ACTION (nav_next) NEXT FIELD NEXT -- the 'next' toolbar option moves the
                                         -- cursor to the next field

ON ACTION (inp_f) -- the action appears when the cursor
                   -- is placed to the field f003

# We create a nested input to show that the toolbar buttons
# which have no ON KEY or ON ACTION clauses during the input
# become hidden. You will see only 2 toolbar buttons
INPUT val5 FROM f005
  BEFORE INPUT
    DISPLAY "*" TO b001
    DISPLAY "*" TO b002
  ON KEY (CONTROL-E)
    LET answer = fgl_message_box("Exit",
      "Do you want to exit the demo?", 4)
    IF answer = 1 THEN
      EXIT PROGRAM
    ELSE CONTINUE INPUT
    END IF

  ON ACTION (back) -- this action appears when the cursor is
                    -- in f004 field
    EXIT INPUT
  END INPUT

# The Restore action (form field b002) sets all toolbar options labels,
# icons and order numbers to their initial values
ON ACTION (restore)
  CALL fgl_dialog_setkeylabel("control-r", "Restart Key", "", 1)
  CALL fgl_dialog_setkeylabel("control-e", "Exit Key", "", 100)
  CALL fgl_dialog_setactionlabel("nav_first", "First", "nav_first.ico", 10)
  CALL fgl_dialog_setactionlabel("nav_last", "Last", "nav_last.ico", 40)
  CALL fgl_dialog_setactionlabel("nav_prev", "Previous", "nav_prev.ico", 20)
  CALL fgl_dialog_setactionlabel("nav_next", "Next", "nav_next.ico", 30)

# The Change_lab action (form field b001) lets the user set new labels, icons,
# and order numbers for toolbar options
ON ACTION (change_lab)
```



```
CALL change_label()
CONTINUE INPUT

END INPUT
END MAIN

#####
# This option changes the label and icon of the specified
# toolbar button
#####

FUNCTION change_label()
DEFINE
    op_label      VARCHAR (15),
    new_label     VARCHAR (15),
    new_icon, info  VARCHAR (250),
    new_order     INTEGER

OPEN WINDOW win2 AT 4,4 WITH FORM "modify" -- opens a new window with a form
    ATTRIBUTES (BORDER)           -- to where the new values are
                                    -- to be entered

OPTIONS FORM LINE 1

DISPLAY "!" TO b001
DISPLAY "!" TO b002
LET new_icon = "" -- the new_icon variable should have at least zero value
                  -- before it is used in functions below

INPUT op_label, new_label, new_icon, new_order FROM f001, f002, f003, f004

ON ACTION (back) -- the Back action (b001 button) sets all the input
                  -- variables to their initial values
    INITIALIZE op_label TO NULL
    INITIALIZE new_label TO NULL
    LET new_icon = ""
    INITIALIZE new_order TO NULL
    EXIT INPUT          -- and terminates input

ON ACTION (choose) -- the Choose action (f003 function field button)
                  -- lets the user chose the new icon using the
                  -- directory tree.
    LET new_icon = fgl_file_dialog ("open", "1", -- The chosen icon should
                                    "Please select a new icon", "defaultpath", -- be previously added to
                                    "my_file", "Icon (ico)| *.ico||")           -- the program
    DISPLAY new_icon TO f003

END INPUT
CLOSE WINDOW win2

# The next part of the source code analyzes the inputted values and applies
# them to corresponding toolbar options
# We use the downshift() function, because user can enter the label values in
# both upper and lower case
    LET op_label = downshift(op_label)
```



```
let info = fgl_dialog_getactionlabel("back")
let info = fgl_dialog_getactionlabel("inp_f")
# In the following CASE statement, the value of the op_label variable is
# compared to the existing labels returned by means of the
# fgl_dialog_getkeylabel()function. The new parameters are set for the option
# the label of which matches the value of the op_label variable

CASE op_label
  WHEN downshift(fgl_dialog_getkeylabel("control-r"))
    CALL fgl_dialog_setkeylabel("control-r", new_label, new_icon, new_order)

  WHEN downshift(fgl_dialog_getkeylabel("control-e"))
    CALL fgl_dialog_setkeylabel("control-e", new_label, new_icon, new_order)

  WHEN downshift(fgl_dialog_getactionlabel("nav_first"))
    CALL fgl_dialog_setactionlabel("nav_first", new_label, new_icon, new_order)

  WHEN downshift(fgl_dialog_getactionlabel("nav_last"))
    CALL fgl_dialog_setactionlabel("nav_last", new_label, new_icon, new_order)

  WHEN downshift(fgl_dialog_getactionlabel("nav_prev"))
    CALL fgl_dialog_setactionlabel("nav_prev", new_label, new_icon, new_order)

  WHEN downshift(fgl_dialog_getactionlabel("nav_next"))
    CALL fgl_dialog_setactionlabel("nav_next", new_label, new_icon, new_order)

  WHEN downshift(fgl_dialog_getactionlabel("inp_f"))
    CALL fgl_dialog_setactionlabel("inp_f", new_label, new_icon, new_order)

  WHEN downshift(fgl_dialog_getactionlabel("back"))
    CALL fgl_dialog_setactionlabel("Back", new_label, new_icon, new_order)
END CASE
END FUNCTION
```

The Form Files

The following form is the form stored in the 'toolbar_form.per' file

```
DATABASE formonly

SCREEN

{
\gp-----q\g
\g|                                |\
\g| \g DEFAULT toolbar\g          \g MODIFIED toolbar\g      |\
\g| \g fields\g                  \g fields\g            |\
\g| [f001           ]           [f003           ]          |\
\g| [f002           ]           [f004           ]          |\
\g|                           [f005           ]          |\
\g|
```



```
\g|  
\g|      [b001           ]     [b002           ]     |\g|  
\g|  
\g|  
\g|  
\g|  
\g|  
\g|-----d\g  
}
```

ATTRIBUTES

```
f001 = formonly.f001, COMMENTS = "Default toolbar displayed";  
f002 = formonly.f002, COMMENTS = "Default toolbar displayed";  
f003 = formonly.f003,  
      -- these keys get the icons, new names and exchange  
      -- their places when this field is active  
      KEY control-e = "Quit" ("exit.ico") ORDER 1,  
      KEY control-r = "Restart" ("refresh.ico") ORDER 100,  
  
COMMENTS = "Modified toolbar (variant 1) displayed";  
f004 = formonly.f004,  
      -- the buttons only get the icons and do not  
      -- switch places or names  
      KEY control-e = "Exit Key" ("exit.ico") ORDER 100,  
      KEY control-r = "Restart Key" ("refresh.ico") ORDER 1,  
      ACTION inp_f = "Input Field" ORDER 90, -- this option is visible  
                           -- only when this field is active  
      COMMENTS = "Modified toolbar (variant 2) displayed";  
f005 = formonly.f005,  
      ACTION back = "Back" ORDER 90, -- this option is visible  
                           -- only when this field is active  
      COMMENTS = "Modified toolbar (variant 3) displayed";  
  
b001 = formonly.b001,  
      config = "change_lab {Change Label}", widget = "button";  
b002 = formonly.b002,  
      config = "restore {Restore Labels}", widget = "button";
```

KEYS

```
-- these are two toolbar buttons associated with keys;  
-- they have no icons  
control-r = "Restart Key" ORDER 1  
control-e = "Exit Key" ORDER 100
```

ACTIONS

```
-- these are four toolbar buttons associated with actions;  
-- they have icons  
nav_first = "First" ("nav_first.ico")ORDER 10  
nav_prev = "Previous" ("nav_prev.ico")ORDER 20  
nav_next = "Next" ("nav_next.ico")ORDER 30  
nav_last = "Last" ("nav_last.ico")ORDER 40
```



The form given below is stored in the 'modify.per' form file

```
DATABASE formonly

SCREEN
{
\gp-----q\g
\g|          |\g
\g| \g Option label\g [f001]   |\g
\g|          |\g
\g| \g New label\g [f002]   |\g
\g|          |\g
\g| \g New icon\g [f003]   |\g
\g|          |\g
\g| \g New order\g [f004]   |\g
\g|          |
\g| [b001]     [b002]   |\g
\gb-----d\g
}

ATTRIBUTES
f001 = formonly.f001, REQUIRED;
f002 = formonly.f002, REQUIRED;
f003 = formonly.f003,
    config = "choose.ico choose",
    widget = "field_bmp",
    COMMENTS = "Press the arrow to open the directory tree";
f004 = formonly.f004 TYPE INTEGER, REQUIRED;

b001 = formonly.b001,
    config = "back {Back}", widget = "button";
b002 = formonly.b002,
    config = "ACCEPT {Change Option}", widget = "button";
```

Image Files

Here are the icons used in the program. If you save them and import into your project, they will be used as the toolbar button labels:



nav_first.ico



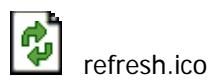
nav_last.ico



nav_next.ico



nav_prev.ico



refresh.ico



exit.ico



choose.ico



Running Commands

4GL applications can be run not directly, but from within another running 4GL application as its child process. You can also run various commands from within 4GL applications.

Parent and Child Processes

If one 4GL application (or any other application invoked using a command) is run from within another 4GL application, such application is called a child process. The application from which a child process is invoked is called a parent process. As a rule, when a child process starts executing, the parent process becomes suspended and 4GL waits until the execution of the child process finishes. After that the execution of the parent application resumes.

The RUN statement is used to specify a command line for the operating system to execute.

The RUN Statement

The RUN statement is used to invoke a command for the operating system or another 4GL application. It has the following syntax:

```
RUN command [WITHOUT WAITING | RETURNING variable]
```

The command can be either a quoted character string specifying the command or the application to run, or a variable of character data type (which can also be a record member or an array element). When the command is invoked, the application which contains the corresponding RUN statement is suspended and resumes its execution only when the command execution is terminated.

Below is an example of invoking a MS Notepad application by means of the RUN statement:

```
RUN "notepad.exe" -- the notepad will open and the program will  
-- be suspended until you close the notepad
```

Child 4GL Applications

Only compiled applications can be invoked like child applications. When you want to call another 4GL application from the parent one, you need to specify its name and the path, if required, as the RUN command. The program name must include the extension of an executable file (exe, 4gi, etc.). If the child program is located in the same project, it is enough to specify only its name. If the application is located in another project, you need to specify the path to its compiled program file.

The child and parent applications must have their own MAIN sections and be independent programs. They cannot share variables. Each program must be independently terminated either by the END MAIN or EXIT PROGRAM statement. Thus calling a child process is different from calling a function.

In Character Mode

If the parent application is run in character mode, using the RUN statement will launch the child 4GL application in the character mode too. None of the applications must be deployed to the application server in this case and if they are located in the same project, you should only specify the name of the application, e.g.:



```
RUN "my_prog.exe"
```

This code will launch the specified program provided that it was compiled.

Child 4GL in the GUI Mode

If the parent application is run in GUI mode, the child application will also be run in GUI mode using the same thin client.

In this case the child application must be deployed to the same application server on which the parent application is running. The child application must be deployed to this application server before it can be launched by the parent application. The parent application is deployed to the application server automatically, when it is run, if you use default settings of Lycia Studio. If you haven't run the child application separately, you have to use the **Deploy to Server** option from the context menu of the program in Lycia Studio to deploy it manually or copy the compiled program to the */progs* directory of your application server. After that you need to specify the name of the program in the RUN statement; if the program is deployed in some subfolder on the application server, you should specify also the path including that subfolder.

If the child application hasn't been deployed to server automatically and you haven't deployed it manually, a runtime error will occur as soon as the parent program encounters the RUN statement, if the application is run in GUI mode.

	Note: It is not advisable to use the RUN statement to run non-4GL applications in the GUI mode.
--	--

The WITHOUT WAITING Keywords

However, this behaviour can be changed so that the parent application did not wait for the child application to terminate. If you specify the WITHOUT WAITING keywords, the execution of the child process will continue along with the execution of the parent application, which will not be suspended.

If the WITHOUT WAITING keywords are in effect, in character mode, the child application is run in background and does not affect the visual display. These keywords are useful when the parent application does not need the result of the execution of the child process to continue the execution. Below is an example of another 4GL application run as a child process, the relative path to the application is specified in the "command":

```
RUN "/progs/utilites/ut.4gi" WITHOUT WAITING
```



Note: The WITHOUT WAITING keywords cannot appear in the RUN statement which contains the RETURNING clause; because this clause means that the result of the child program execution is required by the parent program.

Running Non-4GL Applications in GUI mode

You can run non-4GL applications from a parent application running in character mode by using the RUN statement.

However, it is not advisable to use the RUN statement to run non-4GL applications in the GUI mode. If the child application is a non-4GL one, it will be run in the context of non-interactive desktop and won't be visible in the foreground. Moreover, it will block the execution of the parent program unless the child program is terminated (unless the WITHOUT WAITING keywords described below are used).

This mode of non-4GL programs execution can be convenient when you want to launch a process that does not need user interaction.

However, there is a way to make Windows processes available for the user. This is using the *winexec()* and *winshellexec()* functions.

Using Built-In Functions to Run Windows Processes

Querix 4GL supports several functions to make the program run and execute non-4GL applications so that they are available for the user.

The *winexec()* and *winexecwait()* functions are similar in their effect to the RUN statement, but they make the GUI client execute a program which is on a remote station. It means you can launch any program installed on the computer where the application server is situated.

The *winexec()* function is similar to the RUN...WITHOUT WAITING statement, and the *winexecwait()* function is similar to a simple RUN statement. When these functions are executed in the character mode, they are processed just as the corresponding RUN statements.

The *winexec()* and *winexecwait()* functions can take only one argument which is represented by a path and a name of the executable file (with extension) taken into quotes, or by a variable containing these data. If you want to run a standard program of your system (for example, Notepad, MsPaint, etc., you can omit the path and the extension).

If you use the *winexec()* or *winexecwait()* functions to launch a 4GL application, this application will be run in character mode, even if the parent program is run in GUI mode

Below you can see a simple example of a *winexec()* function application, where the program starts *Skype.exe* if the user enters the letter *s* and starts Microsoft Paint, if the user presses *p*:

```
MAIN
DEFINE a CHAR
WHILE TRUE
    PROMPT "PRESS \"S\" TO START SKYPE AND \"P\" TO START PAINT" FOR CHAR a
```



```

CASE
    WHEN a = "s" CALL winexec( "C:/Program Files/Skype/Skype.exe" )
        MESSAGE "Skype is run and executed successfully"
    WHEN a ="p" CALL winexecwait ("MSpaint")
        MESSAGE "Paint is run and executed successfully"
    OTHERWISE
        MESSAGE "You quit"
        EXIT WHILE
    END CASE
END WHILE
END MAIN

```

Using Built-In Functions to Open Files

There are also built-in functions that can open different files (text files, image files, audio files, etc.) stored in the computer. These are the *winshellexec()* and *winshellexecwait()* functions. The *winshellexec()* function opens the specified file and the program continues its execution. The *winshellexecwait()* function opens the specified file and makes the program wait until the user closes this document; only after that the program execution is resumed.

The functions can take only one argument which is a quoted character string containing the file name and extension and the path to the file. The argument can also be a variable containing this information.
 It is very convenient to use this function with the *fgl_file_dialog()* function which allows the user to chose the file to be opened.

In the following example, when the application is run, the user can press the Control-F key to call the dialog box and to open a file or any other key to quit. When the file is opened, the program will wait for the user to close it and only then end its execution.

```

MAIN
DEFINE a CHAR,
    path VARCHAR(300)
PROMPT "Press F to open a file and any other key to quit" FOR a

IF a Matches "[Ff]" THEN

    LET path = fgl_file_dialog ("Open", "1",
                                "Please select a file to open", "defaultpath",
                                "file.format", "any format|*.*||")
    CALL winshellexecwait(path)

ELSE
    EXIT PROGRAM
END IF

END MAIN

```

The *winshellexec()* and *winshellexecwait()* functions can also be used to run applications. If they are, they perform just the same as the *winexec()* and *winexecwait()* functions. The only difference between these function execution is that *winshellexec[wait]()* is executed through a system shell.



Termination Code

If the child process is a 4GL application, it may return some values to the parent process. To receive these values the parent process should have the RUN statement with the RETURNING clause and the child application should have the EXIT PROGRAM statement specifying the returned value.

Values Returned by the EXIT PROGRAM Statement

We have already discussed the usage of the EXIT PROGRAM statement with regard to a single program. It has somewhat wider syntax when it is used in a child 4GL program though. The syntax of the EXIT PROGRAM statement within a child 4GL application must be as follows:

```
EXIT PROGRAM (exit code)
```

The exit code can be an integer expression, a variable of INTEGER or SMALLINT data type, or a literal integer.

The EXIT PROGRAM statement in a child application performs the same function as in any other position - it terminates the program and no subsequent statements are executed. However, it sends the exit code specified after it to the parent application. The exit code is zero when the program is terminated normally and not due to some error:

```
EXIT PROGRAM (0)
```

The exit code different from zero is usually an integer less than 256 and it specifies the error which caused the child application to terminate. This exit code is passed to the RETURNING clause of the RUN statement.

The RETURNING Clause

The RETURNING clause is used to save the termination code of the child application invoked by the RUN statement. It has a simple syntax:

```
RETURNING variable
```

The variable here is an INTEGER or SMALLINTEGER variable which stores the status code returned to the parent process. This variable can then be examined within the parent program to determine what the result of the child process execution was and what actions should be taken next. The status code is an integer. A zero status code returned means that the child process terminated normally. Non-zero status codes usually specify the error which caused the execution terminate prematurely.

The RETURNING clause can only be used, if the child process is a 4GL program which contains the EXIT PROGRAM statement. When such program completes execution, it returns two bytes of information to the RETURNING clause variable:

- The low byte contains the termination status of the RUN command. The value of this status can be retrieved by calculating the integer variable of the RETURNING clause modulo 256.
- The high byte contains the low byte from the EXIT PROGRAM statement of the child application. Its value can be retrieved by dividing the RETURNING clause variable by 256.

The code below represents a sample 4GL program to be launched as a child process:

```
MAIN
```



```
DEFINE exit_code INT
LET exit_code = 5
EXIT PROGRAM (exit_code)
END MAIN
```

The following program is the parent program containing the RUN statement which calls the one from the example above. The "child_prog" variable here is used to store the name of the child application:

```
DEFINE expg_code, status_code, returned_int INT,
child_prog CHAR(30)

.
.
RUN child_prog RETURNING returned_int
LET status_code = (returned_int MOD 256)
IF status_code <> 0 THEN
    MESSAGE "Unable to run the ", child_prog , " program."
END IF
LET expg_code = (returned_int/256) --will be 0, if no errors occur
DISPLAY " Code from the ", child_prog , " program is ", expg_code
```

Unless an error or signal terminates the child program before the EXIT PROGRAM statement is encountered, the displayed value of *expg_code* is 5. Exercise caution in interpreting the integer variable, however, because under some circumstances the quotient (variable)/256 might not be the actual status code value that the command line returned.

If the child program is terminated by the Interrupt key, the returned value is 256, if it is terminated by the Quit key, the returned value is (3*256), or 758. If the child program is designed in such a way that it can be terminated only by a user, you can include several EXIT PROGRAM (number) statements with different number values. Examination of the returned value in the parent program would then indicate which of the EXIT statements was executed.

Application Launcher Menu

Querix 4GL supports applications which allow to run multiple cascading programs (non-4GL as well as 4GL) and SQL functions simultaneously from a single menu. Application launcher menu is a tool which can be used only in GUI mode. Moreover, it is accessible to the user only if the MDI mode is activated. This is a necessary condition which results from the fact that several applications can be run at once. To activate the MDI mode, add the following line to a script file used by your application:

```
Default.MDImode: true
```

To create an application launcher menu, you have to use a set of built-in functions which are described below.

Specifying the Menu Server

Before the menu can be created and used, you have first to specify which menu server the application should use. To do it, you should call the *set_menu_server()* function. This is an obligatory one, if you don't call it, the menu application won't function. The function has the following syntax:

```
set_menu_server(port, working directory)
```



Here, *port* stands for a VARCHAR variable or a quoted character string specifying the port of the menu server. *Working_directory* is a VARCHAR variable or a quoted character string specifying the directory of the menu server.

You should not confuse the menu server with the application server. If you are not sure which settings to use, you can call the *set_menu_server()* function the following way:

```
CALL set_menu_server(6001, $ServerDir/menu/6001)
```

By default, the application server directory on Windows 7 is: C:/ProgramData/Querix/Lycia

Specifying an Application Launcher Menu

After you have indicated the menu server, you can start creating the application launcher menu. The first step is specifying the menu id. The program will later use this id to reference the menu for adding options and submenus. To specify an application launcher menu id, you should use the *create_menu()* function. The function does not require any arguments. When called, it generates an integer value for indicating the menu id. You can use the function in the following way:

```
DEFINE main_menu_id INTEGER
...
LET main_menu_id = create_menu()
```

Populating the Launcher Menu with Options

As it was mentioned above, the application launcher menu can be populated with options and submenus. Moreover, each submenu can have other nested submenus with their own options. All the objects are placed on the application launcher menu bar in the order in which they are specified in the program code.

To create an option within a menu created by the *create_menu()* function you should use the *menu_add_option()* function. The function syntax is as follows:

```
menu_add_option(menu_id, label, action_id [, enabled])
```

As you can see, the function can take four arguments and only one of them is optional.

The *menu_id* stands for the variable of INTEGER data type which refers to the previously generated id of the menu to which you want the option to be added.

The *label* is a VARCHAR variable or a quoted string specifying the name of the menu option.

The *action_id* stands for an INTEGER variable or value specifying the id by which the program will reference the option. The *action_id* can have any value, but the action ids used in one program should not match. However, an action id can match the menu id.

The *enabled* argument stands for an INTEGER variable or value which indicates whether the menu option will be visible or not. By default, the option is visible, and this argument can be omitted if you don't want to hide the option. The *enabled* attribute can be represented by one of the two values: 1 - which means that the menu option should be visible, and 0 - which will make the program hide the option.

In the following example, the *menu_add_option()* is used to add a visible option to the menu:



```
DEFINE main_menu_id INTEGER  
  
CALL set_menu_server("6001", "C:/ProgramData/Querix/Lycia/menu/6001")  
  
LET main_menu_id = create_menu()  
  
CALL menu_add_option(main_menu_id, "Exit", 1, 1)
```

When, during the application execution, the user presses the Exit option on the menu, the menu will return the integer value 1, which is the action id for this option.

Adding a Submenu

A menu can also have one or several submenus. To add a submenu to your application launcher menu, use the *menu_add_submenu()* function. The function has the following syntax:

```
menu_add_submenu(menu_id, menu_label [, enabled])
```

The arguments here are similar to those of the *menu_add_option()* function. When the function is executed, it returns an integer value indicating the submenu id:

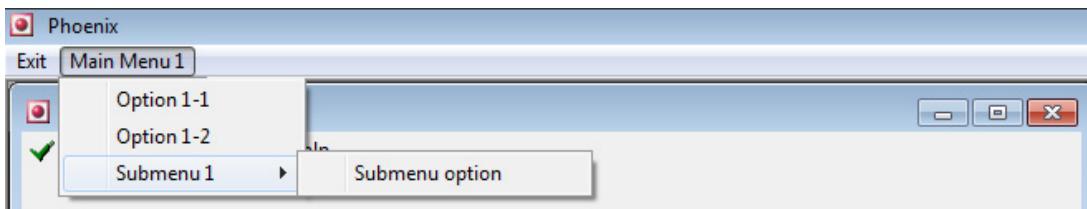
```
DEFINE submenu_id  
  
LET submenu_id = menu_add_submenu(main_menu_id, "Main Menu 1")  
  
CALL menu_add_option (submenu_id, "Option 1-1", 2)  
  
CALL menu_add_option (submenu_id, "Option 1-2", 3)
```

The submenu created by this snippet of the source code will have two options, which have action ids 2 and 3. Note, that to add an option to the submenu, you have to use the submenu id rather than the main menu id, but the action_id should be unique throughout the whole menu regardless whether it is for a submenu option or for the option in the main menu.

You can also add a nested submenu to this submenu. To do this, use the submenu id in the *menu_add_submenu()* function:

```
LET submenu2_id = menu_add_submenu(submenu_id, "Submenu 1")  
  
CALL menu_add_option(submenu2_id, "Submenu option", 4)
```

Therefore, at the current stage, our submenu would look as follows (provided that we add the functions that display the menu; these functions are described below):





Publishing and Executing the Menu

The menu won't be available to the user unless you use the *menu_publish()* function to publish it. The *menu_publish()* function is to be specified after the declarations of all the menu options and submenus your menu should contain. The menu cannot be modified after this function is specified.

The *menu_publish()* function does not need any arguments. It references the menu that is specified before this function.

After you publish the menu, you have to execute it by means of the *execute_menu()* function. The *execute_menu()* function monitors the state of the menu and returns the *action_id* of the option activated by the user. You can use the returned value in different conditional statements in order to specify different program actions for different menu options:

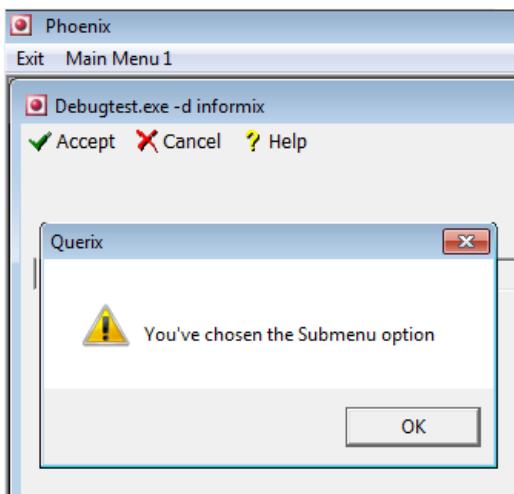
```
CALL menu_publish()

WHILE (TRUE)
LET action_id = execute_menu() -- the action_id gets the value as soon
-- as the user chooses a menu option.

CASE action_id -- specifies different action for different action ids
    WHEN 1 CALL fgl_message_box ("You've chosen the EXIT option")
        EXIT PROGRAM
    WHEN 2 CALL fgl_message_box ("You've chosen Option 1-1 ")
    WHEN 3 CALL fgl_message_box ("You've chosen Option 1-2 ")
    WHEN 4 CALL fgl_message_box ("You've chosen the Submenu Option")

END CASE
END WHILE
```

Now your simple application launcher menu can be displayed and used by the user and the program. When the application is run, and one of the menu options is chosen, you will see the following in the screen:





Executing Programs

One of the most important features of the Application Launcher Menu is the ability to launch several applications at once. There are two functions used to run applications from the application launcher menu.

The *exec_program()* function runs a 4GL program on a GUI server. This function can be invoked in a conditional loop after the program has returned an action id from the corresponding menu option. The function needs four arguments:

```
exec_program(command, server, port, user_id)
```

Each argument is represented by a quoted string or a VARCHAR variable which indicates:

- In the *command* attribute - the path to the program that is to be launched and its name
- In the *server* attribute - the IP or the domain name of the application server
- In the *port* attribute - the port used by the application server
- In the *user_id* attribute - the user account for which the program will be run

Therefore, let's add another submenu containing a menu option which runs a 4GL program:

```
...
LET submenu_id = menu_add_submenu(main_menu_id, "Main Menu 2")
CALL menu_add_option (submenu_id, "4GL program", 5)
...
CASE action_id
  ...
    WHEN 5 CALL fgl_message_box ("A 4GL program will be run")
          CALL exec_program("Hello1.exe", "localhost", "1789", "SYSTEM")
  ...
...
```

You should remember, that any options or submenus specifications you want to add to an existing application launcher menu should be placed between the corresponding *create_menu()* and *menu_publish()* functions.

The *exec_local()* function runs a local application, such as notepad on a Windows system. The function, like the *exec_program()*, is invoked when the user invokes an option of the application launcher menu. The function syntax is:

```
exec_local(command, user_id)
```

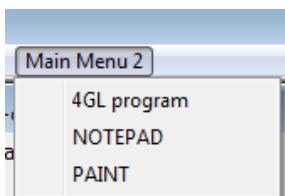
Where *command* stands for a VARCHAR variable or a quoted string specifying the path to the program to be launched. The path can consist of only the executable file name if the launched program is one of the standard Windows programs (Paint, Notepad, etc):

```
CALL menu_add_option (submenu_id, "NOTEPAD", 6)
CALL menu_add_option (submenu_id, "PAINT", 7)
...
CASE action_id
  ...
    WHEN 6 CALL fgl_message_box ("Windows Notepad will be run")
          CALL exec_local("notepad.exe", "SYSTEM") --runs the MS Notepad
```



```
WHEN 7 CALL fgl_message_box ("Windows Paint will be run")
      CALL exec_local("mspaint.exe", "SYSTEM") -- runs the MS Paint
END CASE
```

After you add the second submenu with options "4GL program", "NOTEPAD", and "PAINT", you will have them in your application launcher menu:



The Logic of the Application Menu Performance

The application menu, in fact, imitates the keypress of the key which is specified as the menu option id. It means, that if you specify the option id as 1, the actions associated with this menu option can also be invoked by pressing key 1 on the keyboard.

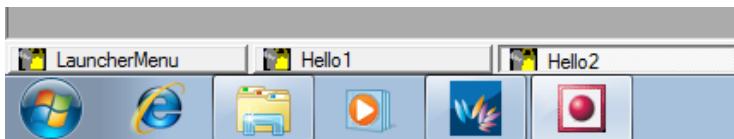
And, vice versa, if you chose the menu option during the input, the option will pass its id to the input field instead of invoking any actions.

Therefore, it is advisable, that the parent program which calls and displays the application launcher menu, doesn't involve any input processes, or launches them only as the result of a menu option activation.

You can create a menu option which starts several applications, for example:

```
WHEN 5 CALL exec_program("Hello1.exe", "localhost", "1789", "SYSTEM")
      CALL exec_program("Hello2.exe", "localhost", "1789", "SYSTEM")
```

When the option with the id "5" is chosen by the user, the program starts two 4GL applications, each in its own window. The window of the application which is run the last ("Hello2.exe") will overlap the window of the first application ("Hello1.exe"). However, you can easily drag the top window to another part of the screen and switch to the bottom one. You can also switch between the launched programs (including the parent and the child applications) using the tabs at the bottom of the screen:



The main difference between the application launcher menu and the RUN statement is that it is possible to switch between the launched programs and to work simultaneously with them. You don't need to close the most recent application to get the control over the previous one.

Although it is not possible to use the application launcher menu within the child application (because its source code usually has no reference to the launcher menu of the parent application), you can always switch



to the parent program and invoke any other programs or processes you need. You can even launch one and the same application for several times, if needed.

Using Scripts to Change the Display Mode

By default, the applications are run in the SDI(Single Document Interface) mode. The *mdimode: true* property makes the program run your application within an MDI (Multiple Document Interface) container. When the application is run in the MDI mode, client automatically maximizes the application container. The size of the container can be modified with the *framewidth* and *frameheight* properties which accept numeric values.

The MDI mode is useful when your application uses the application launcher menu so that several applications are opened in one container.

framewidth

The framewidth property determines the width of the MDI frame in pixels. This will only be referenced if you have default.mdimode set to true.

frameheight

The frameheight property determines the height of the MDI frame in pixels. This will only be referenced if you have default.mdimode set to true.

Flatmdimode

The *flatmdimode: true* property makes all applications be maximized within the MDI container. This may be useful when you don't want to allow the access to the parent window when the user is working with the current one. The *flatmdimode* property can be specified only after the *mdimode* property is specified.

Example

This example consists of several programs which include parent and child programs and the program which launches them all from the application launcher menu. Thus be careful and place all the files containing MAIN sections in different programs, but within the same project.

You will also need to make some changes in the code and some additional preparations before this example will work:

- You need to create a text file and an image file (or any other files you want to be opened via the application launcher menu) which will then be referenced in the code.
- You will need to add the path to some objects which are launched from the application launcher menu. These paths may vary from user to user, so the paths included into this example may appear wrong on your system. See the source code comments for the main program to find out which paths need to be replaced
- You should compile all the four applications
- Then you should deploy some of them manually to the application server. It is required because automatic deployment is enabled only for the programs you launch manually from Lycia Studio and



not for those which are launched indirectly from another 4GL program. To deploy a compiled program manually, right-click it and select "Deploy to Server..." option from the context menu.

The Main Program

This program consists of the 4GL file and the script file which activates the MDI mode used for the application launcher menu. This program can have any name.

```
#####
# This example illustrates the creation and usage of the application launcher
# menu. This is the first program where the menu is created, and from which
# other standard applications and 4GL programs are invoked
#####

MAIN
DEFINE
    main_menu_id,          -- This is the main menu id that
                           -- will be referenced by the
                           -- functions to add options
                           -- and submenus
    submenu1_id,           -- These variables will store the ids of submenus
    submenu2_id,
    action_id      INT   -- And this variable will hold
                           -- the id of the menu option
                           -- activated by the user

# This application includes MDI which can be used only in GUI mode
IF fgl_fglgui()= 0 THEN
    CALL fgl_winmessage( "Wrong mode",
                         "Run this application in the GUI mode." ||
                         "\nThe MDI mode required for application" ||
                         "\nlauncher menu cannot be run in character" ||
                         "\nmode.", "info")
    EXIT PROGRAM
END IF

#####
# Setting the menu server and creating the menu
#####

# First, we must set up the menu server by specifying its port and working
# directory. The menu server on Windows 7 is usually created
# in the directory given below regardless of the directory
# to which you install Lycia. On Windows XP it is located in
# C:/ProgramFiles/Querix/Lycia/menu/6001. Make sure you specified
# the correct location.
CALL set_menu_server(6001, "C:/ProgramData/Querix/Lycia/menu/6001")

# When we create the menu, this function returns the menu ID
# which we assign to a variable and will then use when
# creating menu options
LET main_menu_id = create_menu()
```



```
# This is the first of the submenus we are going to add to the main menu.
LET submenu1_id =
    menu_add_submenu(main_menu_id, -- variable to store sub-menu ID
                    "Standard_Programs", -- menu ID to which it will belong
                    1)                      -- sub-menu name
                                         -- action ID to reflect the user's choice

# Then we create four options for this submenu
CALL menu_add_option(submenu1_id, -- the id of the submenu to which it
belong
                      "Notepad", -- option name
                      2)          -- action ID to reflect the user's choice
CALL menu_add_option(submenu1_id, "Paint", 3)
CALL menu_add_option(submenu1_id, "Calculator", 4)
CALL menu_add_option(submenu1_id, "E-mail", 5)

# The following two options are added directly to the main menu. The syntax is
# the same as that used for adding submenu options, with the only exception that
# the main_menu_id is specified as the first argument here
CALL menu_add_option(main_menu_id, "4GL_Program(Character Mode)", 6)
CALL menu_add_option(main_menu_id, "4GL_Program(GUI Mode)", 7)

# Another submenu with two options is added to the main menu
LET submenu2_id = menu_add_submenu(main_menu_id, "Open_Documents", 8)
CALL menu_add_option(submenu2_id, "Document_1", 9)
CALL menu_add_option(submenu2_id, "Document_2", 10)

# The last option is added
CALL menu_add_option(main_menu_id, "Exit", 11)

#####
# Publishing and using the menu
#####

# After the menu has been created and all the submenus and options have been
# added to it, we must execute the menu_publish() function. Otherwise, the menu
# will not be available. Now we cannot modify the menu in any way
CALL menu_publish()

# To keep the menu on the screen until the user presses the Exit or X buttons,
# the rest of the code is enclosed within the WHILE loop
WHILE TRUE

# Next, we call the execute_menu() function. It returns the id of the menu
# option activated by the user. This id is then passed to the CASE statement and
# the corresponding action is executed
LET action_id = execute_menu()

CASE action_id

WHEN 2
```



```
CALL winexec( "notepad" )                                -- These options in the first submenu
WHEN 3                                                 -- call standard programs: "Notepad",
CALL exec_local( "MSPaint" , "SYSTEM" )                  -- "MSPaint""Calculator" and the default
                                                        -- mail client.
                                                        -- As the programs are standard, you don't
                                                        -- have to specify the path to their .exe
                                                        -- files or the extension itself.
WHEN 4                                                 -- If you run this application on Linux,
CALL exec_local( "calc" , "SYSTEM" )                      -- you should replace them with default
                                                        -- Linux applications
WHEN 5
CALL winshellexec( "mailto:
reciever@mail.com?cc=copy@mail.com&subject=Subject" )

# Below is the pattern for using the "mailto:" command
# mailto:<address_of_the_receiver>?cc=<whom_a_copy>
# &bcc=<whom_a_hidden_copy>&subject=<subject_of_the_message>
# &body=<text_of_the_message>"path_to_the_attached_file" .

# When the user selects the following option, a 4GL program is executed in the
# character mode with the help of the winexec() function. The invoked program
# itself runs a standard application and a 4GL program, and needs a separate
# window too. You must specify the path to its location on the file system,
# normally it would be your output folder.
WHEN 6
CALL winexec ( "C:/Users/felvis/workspace/run/output/run_comm.exe" )

# This option, when activated, runs a 4GL program in the GUI mode.
# To be able to launch the program you need to deploy it
# to the application server first. It must be the same application
# server on which the main program runs. You can change the host and port
# according to your needs.
WHEN 7
CALL exec_program ( "gui_prog.exe" , "localhost" , "1789" , "SYSTEM" )

# The following two options are used for opening documents on the local system.
# These documents must exist on your OS and you must specify the correct
# paths to them as the arguments. The paths below are given only for example

WHEN 9
CALL winshellexecwait( "C:/Users/felvis/workspace/run/source/1.txt" )
WHEN 10
CALL winshellexec ( "C:/Users/felvis/workspace/run/source/1.jpg" )

WHEN 11
    EXIT PROGRAM
END CASE
END WHILE

END MAIN
```



The Script File

This script file should be added to the requirements of the main program to enable the MDI mode.

```
# We set the MDI mode for the application
default.MDImode:true
# This maximizes the GUI-run 4GL application within the MDI frame
*flatmdimode:true
# The width of the MDI frame is set
*framewidth:675
# The height of the MDI frame is set
*frameheight:690
```

The Program Launched in GUI Mode

The program is called "gui_prog.exe" and must be located at the same project as the main program. It must be compiled and deployed to the GUI server manually before it can be used in the application launcher menu.

It consists of a .4gl file and a form file. Here is the 4GL file:

```
#####
# This is the source file of the 4GL program "gui_prog" run in the GUI mode from
# the application launcher menu when the corresponding menu option is selected
#####

MAIN
DEFINE f_name,
      l_name   VARCHAR (30),
      answer   CHAR(3)

OPTIONS
ACCEPT KEY F3

OPEN WINDOW win1 AT 3,5 WITH FORM "pers_form" ATTRIBUTES (BORDER)

DISPLAY "!" TO accept_but
DISPLAY "!" TO quit_but

INPUT BY NAME f_name,l_name

ON KEY (F3)

CASE
WHEN f_name IS NULL AND l_name IS NULL
  CALL fgl_winmessage("WARNING",
                      "You should provide all the information about yourself",
                      "exclamation")
CONTINUE INPUT

OTHERWISE
  CALL fgl_winmessage("DATA SAVED",
                      "Thank you! Your personal data was saved.",
```



```
        "info")
    EXIT INPUT
END CASE

ON KEY (F2)
LET answer = fgl_winqestion("QUIT?", "Do you really want to quit?",
    "Yes", "Yes|No", "question", 1)

IF answer = "Yes" THEN
    EXIT PROGRAM
ELSE
    NEXT FIELD f_name
    CONTINUE INPUT
END IF

END INPUT

CLOSE WINDOW win1

END MAIN
```

The Form File

The form file included into gui_prog program is called "pers_form.per".

```
DATABASE formonly

SCREEN {

\gp-----q\g
\g| [f01 ] [f02 ] | \g
\g| [f03 ] [f04 ] | \g
\g| [f06 ] [f05 ] | \g
\gb-----d\g

}

ATTRIBUTES

f01 = formonly.lab1, widget="label", config = "First Name:", CENTER, COLOR =
BLUE BOLD;
f02 = formonly.f_name TYPE VARCHAR;
f03 = formonly.lab2, widget="label", config ="Last Name:", CENTER, COLOR = BLUE
BOLD;
f04 = formonly.l_name TYPE VARCHAR;
f05 = formonly.quit_but, widget = "button", config="F2 Quit", CENTER, COLOR = RED
BOLD REVERSE;
f06 = formonly.accept_but, widget= "button", config="F3 Accept", CENTER,
COLOR=GREEN BOLD REVERSE;

INSTRUCTIONS
```



DELIMITERS "[]"

The Parent Program Launched in Character Mode

This example includes two programs which are launched in the character mode. The parent program is launched from the application launcher menu. The child program is launched from within the parent program. You don't need to deploy the parent program, because it is launched in character mode, but you need to compile it before it can be used. You should also replace the path to it specified in the main program to reflect the actual location of the compiled program file in your workspace.

Here is the source code of the parent program called "run_comm.exe":

```
#####
# This is the 4GL program "run_comm" which is launched in the character mode
# when the corresponding button in the application launcher menu is pressed. It
# itself runs a child 4GL application (ext_prog.exe)
#####

MAIN
DEFINE
    str_command CHAR(256), -- the variable used to store the command
    ret_int      INTEGER      -- the variable to be used for analyzing the value
                                -- returned by a child program

OPTIONS
MESSAGE LINE LAST - 1

DISPLAY "The parent program is being executed, press any key to continue"
AT 1,2 ATTRIBUTE(GREEN, BOLD)
CALL fgl_getkey()

DISPLAY "The RUN invokes notepad text editor: RUN ""notepad"" " AT 3,2
ATTRIBUTE(GREEN, BOLD)
SLEEP 1
RUN "notepad" -- we use the command directly as the quoted string
                -- in the RUN statement - it can be done only if the program
                -- is run in character mode, otherwise use winexec()

DISPLAY "The program execution is resumed only after the Notepad is closed"
        AT 4,2 ATTRIBUTE(GREEN, BOLD)
MESSAGE "Press any key to continue . . ."
CALL fgl_getkey()
MESSAGE ""

DISPLAY "The RUN statement with the WITHOUT WAITING option invokes notepad:"
        AT 6,2 ATTRIBUTE(GREEN, BOLD)
DISPLAY "RUN ""notepad"" WITHOUT WAITING" AT 7,2 ATTRIBUTE(GREEN, BOLD)
SLEEP 1
LET str_command = "notepad" -- we assign a command to a variable
RUN str_command WITHOUT WAITING -- we use the variable containing the
                                -- command in the RUN statement
                                -- the WITHOUT WAITING keywords allow you to
                                -- continue parent program execution
                                -- without having to close the Notepad
```



```
DISPLAY "The program resumes execution without waiting for Notepad termination."
    AT 8,2 ATTRIBUTE(GREEN, BOLD)
MESSAGE "Press any key to continue . . ."
CALL fgl_getkey()
MESSAGE ""

DISPLAY "Press any key to run child 4GL application ext_prog.exe"
    AT 10,2 ATTRIBUTE(GREEN, BOLD)
CALL fgl_getkey()
# This program must reside in the same project as the parent one,
# so we only specify its name here
RUN "ext_prog.exe" RETURNING ret_int
CLEAR SCREEN
# To find out whether the child program was executed successfully
# we need to analyse the value of the returned variable(ret_int)

DISPLAY "The child program was terminated, we returned to the parent program."
    AT 3,5 ATTRIBUTE(GREEN, BOLD)
DISPLAY "The value returned is ret_int = ", ret_int USING "<<<&"
    AT 5,5 ATTRIBUTE(YELLOW, BOLD)
MESSAGE "Press any key to finish . . ."
CALL fgl_getkey()

END MAIN
```

The Child Program Launched From within a Parent Program

The child program called "ext_prog.exe" should be located in the same project as the parent program "run_comm". If they are not, you must specify the path to the program in run_comm source code. If they are located in the same project, the path is not required. You need to compile the child program, but you don't need to deploy it to the application server, as it will be run in character mode.

Here is the source code of the child program:

```
#####
# This is the child 4GL program "ext_prog" which sends the return code to
# the parent program (run_comm.exe)
#####

MAIN
DEFINE ret_int INTEGER

OPEN WINDOW ext_win AT 2,2 WITH 15 ROWS, 70 COLUMNS ATTRIBUTE(BORDER)
DISPLAY "This is the child program called ext_prog. It assigns a value"
    AT 2,2 ATTRIBUTE (GREEN, BOLD)
DISPLAY "to a variable and returns it to the parent program."
    AT 3,2 ATTRIBUTE (GREEN, BOLD)
LET ret_int = fgl_winprompt(7,10,
    "Enter the integer number you want to return" ||
    "\nto the parent program: ", "7", 10, 2)

CLOSE WINDOW ext_win
EXIT PROGRAM (ret_int) -- the return code is specified in the parentheses
```



END MAIN



WORK WITH DATABASES

This section will teach you to use 4GL applications to access and modify a database. In order to be able to use the applications this section contains you need to have your Lycia installed together with a graphical client. You will also need an access to any database and the client of this database installed on your machine together with Lycia. You will need to establish database connection for Lycia. This process is out of the scope of this guide, you can find the necessary instructions in other Lycia documentation.



Databases in 4GL

The 4GL language includes embedded SQL used to work with databases. The 4GL statements discussed so far and the form files are typically used to process and display data retrieved from a database. To retrieve some data from a database and to add some data to it, a 4GL program may include some SQL statements either directly in the source code, or embedded with the help of special syntax.

To be able to work with a database you need to connect to it. After you have connected to the database you want to use, you need to instruct your 4GL application which database it should use when compiling and running. The DATABASE statement is used for these purposes.

The DATABASE Statement

The DATABASE statement opens a database. This database is then used as the default database at compile time or as the current database at runtime. The syntax of the DATABASE statement is as follows:

```
DATABASE database_name [EXCLUSIVE]
```

Here the database name is the name of the Informix database you want your program to use. It can be represented by a variable containing the database name, or by the database name itself without quotation marks; the database name should not contain white spaces:

```
DATABASE my_database
```

The database name can contain the server name and the pathname, the syntax differs depending on the database type used:

```
#for IBM Informix Dynamic Server database only
DATABASE database_name@server
DATABASE "//server/database_name"

# for IBM Informix SE database only
DATABASE "/pathname/database_name@server"
DATABASE "//server/pathname/database_name"
```

The EXCLUSIVE keyword, if used, means that the database will be accessible only for the user who opened the database in the exclusive mode. The other users won't be able to use the database at the same time. You can use the EXCLUSIVE keyword only if the DATABASE statement appears within a MAIN or a FUNCTION block. When the database is opened, its exclusive mode cannot be changed. To allow other users to access to the database you must close the database opened in the exclusive mode by means of the CLOSE DATABASE statement. You can also reuse the DATABASE statement without the EXCLUSIVE keyword which will close the database and reopen it in normal mode.

You can use the DATABASE statement in a source file in two ways:

- To specify the default database which is to be used at compile time
- To specify the current database which is to be used at runtime



The Default Database

The default database is declared by the DATABASE keyword preceding any program block within the source file. It must be specified outside any program block at the beginning of the module. If you want to specify the database for all the modules used by the program, you can specify it in the [GLOBALS](#) file, preceding the GLOBALS keyword and then link the file to the modules. Here is an example of a GLOBALS file containing the default database specification:

```
DATABASE qx_db
GLOBALS
DEFINE
...
END GLOBALS
```

The Current Database

The current database is the database to be referenced by the program at runtime. It is required for the SQL statements embedded in 4GL to be able to reference the database. The database remains the current database of the application until the application is terminated, another database is specified as the current one by means of another DATABASE statement, or the database is closed using the CLOSE DATABASE statement. After the database is closed by the CLOSE DATABASE statement, the SQL statements will no longer be able to reference it.

The current database needs not to precede all the other program blocks unlike the default database specification. The DATABASE statement specifying the current database can appear within the MAIN block or within one of the FUNCTION blocks. It must follow any DEFINE statement of that block.

If the DATABASE statement specifying the default database (that is preceding the MAIN block) is present, and no DATABASE statement is used at the beginning of the MAIN block to specify the current one, the DATABASE statement preceding the MAIN block is treated as specifying both the default and the current databases.

The CLOSE DATABASE Statement

The DATABASE statement specifies the database, which functions as the default database at compile time and as the current database at runtime. If you want to use any other database at runtime later in your application, you should use the CLOSE DATABASE statement and then another DATABASE statement following it, which will specify the name of another database. The CLOSE DATABASE statement closes connection to the database used as current at the moment.

If you plan to use another database on the same database server, you can skip the CLOSE DATABASE statement and use the DATABASE statement with the name of the new database instead. It will close the previously opened connection and open the new one. If you want to open a database on another server, you should close the current database explicitly using the CLOSE DATABASE keywords.

Creating and Dropping a Database

The DATABASE statement described above can be used, if the database you reference already exists. Use the CREATE DATABASE statement to create a new database, if you cannot use an existing one.

Though Lycia supports a number of databases, the CREATE DATABASE statement may not work properly in some of them because the database creation is very specific with regard to the database type and cannot be effectively translated. A database can be created using the corresponding native tools, e.g. you can use dbaccess for an Informix database or OCI for Oracle.



	<p>Note: It is advisable that you use native tools for creating a database with which you will then work using Lycia.</p>
---	--

Database Tables

The information in a database is stored in the form of tables. Tables are used to organize data contained in the database using the model of vertical columns and horizontal rows. A table has a specified number of columns, but it can have any number of rows. The columns have their unique names, whereas the rows do not have names. Any information unit has its own position in the database, which is determined by the table it is stored in, the column and the row.

You cannot add information directly to a database; you should add it to a table within that database. Tables can be created, modified, and deleted with the help of the SQL statements embedded into 4GL.

To create a table the CREATE TABLE statement is used. This statement can be used only if the current database is specified. If your program has no DATABASE statement, or of the database is closed before this statement is executed, an error will occur.

It creates a table with the specified columns in the current database. Its syntax is as follows:

```
CREATE TABLE table_name
  (Column_definition, [table constraint])
```

The table name must be unique among the other tables in the used database; it needs not to be quoted, but it cannot contain white spaces. In an ANSI-compliant database the table name must be preceded by the *owner* name (*owner.table_name*) which is the user name of the owner of the table. Thus the table name must be unique among all the other tables of the same owner.

Defining Table Columns

The only required clause of the CREATE TABLE statement is the column definition clause. Here you must specify the names and the data types of the columns you want to include into the table. Each table must contain at least one column. You can optionally place some constraints on a table column and/or specify the default value, but the minimum required are the column name and its data type. This statement has the following syntax:

```
column_name data_type [DEFAULT clause] [column_constraint]
```

The column name must be unique among the other columns of the same table; it needs not to be quoted, but it cannot contain white spaces. The data type is assigned to a column in the same way as to any 4GL variable, you can specify the precision and scale for the corresponding variables.

```
CREATE TABLE my_table(
    first_name CHAR(15),
    last_name CHAR(30),
    age INT
)
```



Data Types Used When Declaring Columns

There are some data types, which can occur in a database, but variables dealing with these columns should normally be declared of the standard 4GL data types. 4GL recognises the following data types used by the database server:

- SERIAL - is an integer data type analogous to the INT data type.
- NCHAR - is a character data type which has the same characteristics as the CHAR data type.
- NVARCHAR - is a character data type which has the same characteristics as the VARCHAR data type.

The above listed types can be applied in the CREATE TABLE when you want to create the columns of these data types.

A column can be assigned any of the 4GL data types and the SERIAL data type. The SERIAL data type is an integer data type which can be used only in SQL statements dealing with database tables. In 4GL the columns of the SERIAL data type can be referenced by variables of the INTEGER data types. The SERIAL columns are generally used for the row indexes. As the data to them is added automatically as soon as any other value is added to a row; they contain the ordinal row number. All the other column types follow the same rules as the variable of the corresponding data type.

Here is an example of a table created. It is called "my_table" and contains four columns one of which is of SERIAL data type:

```
CREATE TABLE my_table(
    serial_column SERIAL,
    first_name CHAR(15),
    last_name CHAR(30),
    age INT
)
```

The DEFAULT Clause

The DEFAULT clause can follow the column name and data type of a column. It is used to specify the default value for the column. This default value is inserted into the column, when no value is specified explicitly. If the DEFAULT value is not specified and the column allows NULL values, the default value is NULL.

The DEFAULT keyword must be followed by a literal value, the NULL or USER keyword. The USER keyword returns the login name of the current user. A column must be of CHAR or VARCHAR data type at least 8 characters long to accept the default USER value. If the column is of the DATETIME data type, it can also be followed by the CURRENT operator with the DATETIME qualifier or without it. The default value of a column of the DATE data type can be represented by the TODAY operator.

Below is an example of the DEFAULT clause in use:

```
CREATE TABLE my_table(
    serial_column SERIAL,
    first_name CHAR(15) DEFAULT USER,
    last_name CHAR(30) DEFAULT NULL,
    registered_on DATE DEFAULT TODAY,
    last_visit DATETIME YEAR TO SECOND DEFAULT CURRENT
)
```



The NOT NULL Constraint

If no default value is specified, the NULL value would be used as the default one, unless you place a not null constraint on the column. If the NOT NULL keywords are following the column name and type and no default value is specified, you must enter the value into that column when filling the row with values. Otherwise an error will occur.

The table below will require some data to be entered into the first column, the second column will contain the default value, if nothing is entered:

```
CREATE TABLE my_table(
    first_name CHAR(15) NOT NULL,
    last_name CHAR(30) DEFAULT "Smith" NOT NULL
)
```

Column-Level Constraints

A column-level constraint is applied only to the column for which it is specified. They impose some restrictions on the values which can be contained by the table column. The constraint should follow the column name and data type and the DEFAULT clause, if there is one. A constraint can be used to:

- Define a column as unique
- Define a column as a primary key
- Define a column which references a column in another table (a foreign key)
- Apply a condition to be met before some values can be inserted into the column

The dependencies between the columns of different tables are created with the help of the foreign and primary keys. You should first create tables which do not depend on other tables and only then create other tables which depend on the previously created ones.

Defining a Column as Unique

To define a column as unique, you should place the UNIQUE keyword at the end of the column definition clause. This constraint means that the column can accept only unique data and you cannot insert duplicate values in it, however, the column may accept NULL values. Thus if the column contains value "100", another "100" cannot be inserted.

Here is an example of the UNIQUE constraint usage:

```
CREATE TABLE my_table(
    user_id CHAR(15) DEFAULT USER NOT NULL UNIQUE,
    e_mail CHAR(30) NOT NULL UNIQUE,
    nickname CHAR(20) UNIQUE
)
```

Defining a Column as a Primary Key

A primary key is a column that contains a non-null unique value for each row in a table. Each table can have only one column with the PRIMARY KEY constraint and this column cannot be marked as UNIQUE. To define a column as a primary key, you need to add the PRIMARY KEY keywords to the end of the column definition clause. The columns of BYTE or TEXT data type cannot be defined as primary key. The primary key columns are generally referenced by foreign key columns from other tables to create interdependencies between the tables.

Below is an example of the PRIMARY KEY constraint in use:



```
CREATE TABLE my_table(
    serial_column SERIAL PRIMARY KEY,
    first_name CHAR(15) DEFAULT USER,
    last_name CHAR(30) DEFAULT NULL
)
```

Defining a Column as a Foreign Key

A foreign key constraint on a column joins and establishes dependencies between tables. A foreign key column must reference a UNIQUE or PRIMARY KEY column in another table. For every entry in the foreign-key columns, a matching entry must exist in the unique or primary key columns specified in the reference. This means that a column declared as a foreign key can accept only those values which are present in the column it references (either unique or primary key). The values in the foreign key column do not need to be unique, but they cannot be different from the values in the referenced column.

	Note: Tables should be created in the following order: first create the tables which do not have the dependencies on other tables (with UNIQUE or PRIMARY keys) and then create the tables with the FOREIGN KEY which depend on the previously created tables.
---	---

To impose a foreign key constraint on a column, you must specify the REFERENCES clause at the end of the column definition which references a unique or primary key column in another table.

The syntax of the REFERENCE clause is as follows:

```
REFERENCES table_name [(column_name)]
```

Here the table name is the table which contains the column you want to reference and the column name is the name of a unique or primary key column in that table. The data types of the foreign key column and the column it references must be the same or compatible.

Below is an example which contains two tables. The first table has a primary key column which is referenced by a foreign key column of another table:

```
CREATE TABLE table_1(
    user_id INT UNIQUE
    user_name CHAR(15) NOT NULL PRIMARY KEY,
)

CREATE TABLE table_2(
    login CHAR(15) REFERENCES table_1 (user_name),
    e_mail CHAR(30)
)
```

If you are referencing a primary key column, you can omit the column name specification, indicating only the table name. Since each table can have only one primary key column, you can specify only "REFERENCE table_name" to create a foreign key. If the REFERENCE clause references a UNIQUE column, the column name should be included.



The CHECK Condition

Check constraints allow you to specify conditions that must be met before data can be assigned to a column. This clause consists of the CHECK keyword and the condition in parentheses. The condition must be a Boolean expression, which accepts [Boolean operators](#), [relational operators](#), and operators [BETWEEN... AND](#), [INQ](#). The usage of the last two operators in 4GL is restricted to the WHERE clause of the COLOR form field attribute, however, their usage in SQL statements is not restricted. At the column level the CHECK clause is placed at the end of the column definition clause. It specifies the range of values which can be inserted into the column, e.g.:

```
CREATE TABLE my_table5(
    order_id INT,
    product_id INT CHECK (product_id BETWEEN 1 AND 999),
    amount DEC
)
```

Table-Level Constraints

Table-level constraints apply to a specified column or a set of columns within the table.

The syntax of table level constraints is similar to that of the column-level constraints and they have the same effect. The main difference between them is that the table level constraints contain the name of the columns to which they apply.

The table-level constraints can be used to achieve the same effects as the column-level ones, but for a set of columns at once:

- To define a column as unique, you need to add the UNIQUE keyword after the column definition clause and the name of the column or columns which you want to make unique, e.g.:

```
CREATE TABLE my_table(
    user_id CHAR(15) DEFAULT USER,
    e_mail CHAR(30) NOT NULL,
    nickname CHAR(20),
    UNIQUE (user_id, e_mail)
)
```

- To define a column as a primary key, you need to specify the PRIMARY KEY keywords after the column definition clause and specify the name of the column name which you want to make primary key in the parentheses. You can specify only one column in this case, e.g.:

```
CREATE TABLE my_table(
    serial_column SERIAL,
    first_name CHAR(15) DEFAULT USER,
    last_name CHAR(30) DEFAULT NULL,
    PRIMARY KEY (serial_column)
)
```

- To define a column as a foreign key you need to specify the FOREIGN KEY keywords together with the column which you want to mark as the foreign key at the end of the column definition clause. This must be followed by the REFERENCE clause. The REFERENCE clause must specify the name of



the table. If it references a primary key column, the column name can be omitted, however, it should be specified, if the referenced column is UNIQUE.

In the example below two columns of the second table have foreign key on the UNIQUE and PRIMARY KEY columns of the first table; the order of the columns in both sets of parentheses must be the same:

```
CREATE TABLE table_1(
    user_id INT UNIQUE
    user_name CHAR(15) NOT NULL PRIMARY KEY,
    user_pwd CHAR(30)
)

CREATE TABLE table_2(
    login CHAR(15) NOT NULL,
    ref_id INT NOT NULL,
    e_mail CHAR(30) PRIMARY KEY,
    FOREIGN KEY (login, ref_id)
        REFERENCES table_1 (user_name, user_id)
)
```

- To apply a condition to be met before some values can be assigned to columns the CHECK clause is used. The syntax is the same as at the column level, but it is placed after the column definition clause. In the example below the two columns specified in the CHECK clause must have the same values:

```
CREATE TABLE my_table5(
    order_id INT,
    product_id INT,
    price_tot MONEY,
    transf_tot MONEY,
    CHECK (price_tot = transf_tot)
)
```

Naming Constraints

When you create a constraint, either table- or column-level one, you may give it a name. If you do not give a name to the constraint, the database server assigns some name to it. However, for your convenience it is better to give names to constraints which you then want to modify.

To give a name to a constraint add the CONSTRAINT keyword after specifying the constraint itself followed by the name you want to assign. The syntax is as follows:

```
CONSTRAINT const_name [ENABLED|DISABLED]
```

The constraint name must be unique among other constraint names. The ENABLED and DISABLED keywords are optional. By default, a constraint is enabled when it is created. A disabled constraint has no effect. The name of the constraint can then be used to enable or disable a constraint in an existing table. This will be discussed in details in the following chapters.

Here is an example of a table created with named constraints:



```
CREATE TABLE table_2(
    login CHAR(15) NOT NULL,
    ref_id INT NOT NULL,
    e_mail CHAR(30) PRIMARY KEY CONSTRAINT pk_e_mail,
    FOREIGN KEY (login, ref_id) REFERENCES (user_name, user_id)
        CONSTRAINT fk_login_ref_id
)
```



Note: You should be careful, because, if the constraint is disabled, and some values are entered to the table which do not meet the requirements of the constraint, enabling it would be impossible.

Temporary Tables

You can create temporary tables. The syntax of a temporary table is similar to that of the regular one and it follows all the rules discussed above which concern the constraints and column definitions:

```
CREATE TEMP TABLE table_name
(Column_definition, [table constraint])
```

The table name of a temporary table must be different from the names of other tables in the database.

A temporary table created explicitly is deleted, when one of the following occurs:

- The application used to create the temporary table terminates.
- The database in which the table is created is closed by the application and another database is opened.

Here is an example of a temporary table created:

```
CREATE TEMP TABLE my_TEMP_table(
    serial_column SERIAL,
    first_name CHAR(15) DEFAULT USER,
    last_name CHAR(30) DEFAULT NULL,
    PRIMARY KEY (serial_column)
)
```

Privileges

In order to allow other users to use a database or a table created by you, you should grant privileges to those users. The GRANT statement is used for that purpose.

Database-Level Privileges

The syntax of the statement used to grant database-level privileges is as follows:

```
GRANT privilege TO user
```



Here the user is either one or more names of the users separated by commas to whom you want to grant the specified privilege, or the PUBLIC keyword, which means that the privilege will be granted to everyone. The “privilege” can be one of the following:

- CONNECT - gives the ability to query and modify data, create temporary tables, grant privileges on a table provided that the user is the owner of that table.
- RESOURCE - gives the ability change the structure of the database, i.e. to create and modify regular tables.
- DBA - gives all database-level privileges of the previous two levels. It also gives the ability to grant DBA privileges to other users.

Here is an example which gives everyone the CONNECT privilege for the current database:

```
GRANT CONNECT TO PUBLIC
```

Table-Level Privileges

It is possible to grant privileges not for the whole database, but for a specific table. The syntax of a table-level GRANT statement is as follows:

```
GRANT privilege ON table_name TO user
```

Here the “table name” is the name of the table you want to grant privileges for and the “user” is the user or users to which the privilege is granted. The “user” can be substituted by the PUBLIC keyword specifying that the privilege is granted to everyone.

The privilege granted on the table-level can be one of the following:

- INSERT - grants the ability to insert rows into the table.
- DELETE - grants the ability to delete rows from the table.
- SELECT - provides the ability to perform queries. You can use SELECT (column_name) format to restrict the query to one or several columns.
- UPDATE - provides the ability to update rows in the table. You can use UPDATE (column_name) format to restrict the update to one or several columns.
- REFERENCES - grants the ability to reference columns in the foreign key constraint.
- INDEX - grants the ability to create a permanent index.
- ALTER - grants the ability to change the table structure.
- ALL - grants all the above mentioned privileges.

Most of the above mentioned activities will be discussed in the next chapters. Here are some examples of the table-level GRANT statement:

```
GRANT ALL ON my_tab TO user123
GRANT SELECT(item_id, item_name) ON goods_tab TO PUBLIC
```



Getting Feedback from the Database

After you have created a database and some tables within it, you may want to know whether they have really been created. It can be done indirectly by placing queries to a specific table, the process of creating queries will be, but this may result in errors, if the table does not exist.

Checking if the Table Already Exists

When compiling, Lycia cannot check whether the table you want to create already exists in the database. If it does, the CREATE TABLE will result in an error at runtime and the program will be terminated. To avoid such situation, the *fgl_find_table()* function can be used.

The function can take only one argument which is the table name represented as a quoted character string or a CHARACTER data type variable.

When the function is invoked, it returns TRUE if the specified table exists in the current database, and FALSE if it does not exist.

The following example demonstrates how this function can be used in practice:

```
DATABASE s_auth
MAIN
DEFINE tablename VARCHAR (30)
PROMPT "Enter the name of the table you want to create" FOR tablename

IF fgl_find_table(tablename) THEN
DISPLAY "The table ", tablename, " exists in s_auth" at 5,5
ELSE
DISPLAY "The table ", tablename, " can be created" at 5,5
END IF
CALL fgl_winmessage("Exit","Press any key to close this demo
application","info")
END MAIN
```

	Note: The <i>fgl_table_info()</i> function can be used only with permanent tables, it cannot be used with temporary tables.
---	--

Retrieving Information about a Column

To return information about a specified database column, you can use the *fgl_column_info()* function. The function takes two arguments, represented by quoted character strings containing the table name and the column name, or by CHARACTER data type variables containing these data.

```
fgl_column_info(table, column)
```



The function returns two INTEGER values which indicate the column type (the first returned value) and the column size (the second returned value). The column *type* is an integer number, corresponding to a data type available in the column. The table of correspondence is given below:



Integer code	Data type	Integer code	Data type
0	CHAR	10	DATETIME
1	SMALLINT	11	BYTE
2	INTEGER	12	TEXT
3	FLOAT	13	VARCHAR
4	SMALLFLOAT	14	INTERVAL
5	DECIMAL	15	NCHAR
6	SERIAL	16	NVARCHAR
7	DATE	45	BOOLEAN
8	MONEY	-1	Column not found

The column *size* is the length of the column fields, i.e., the maximum number of characters that can be inputted /displayed to the column.

As the function returns two values, you should assign these values to a RECORD variable or use the RETURNING clause of the CALL statement when invoking this function. Therefore, the next two variants are possible:

```
DEFINE col_type, col_size INT,
      tname,cname CHAR (100)
LET tname = "users"
LET lname = "username"
CALL fgl_column_info(tname,cname) RETURNING col_type, col_size
DISPLAY "The column size is ", col_info.col_size AT 2,2
DISPLAY "The column type is ", col_info.col_type AT 3,2
```

or

```
DEFINE col_info RECORD
      col_type, col_size INT
      END RECORD

LET col_info = fgl_column_info("users", "username")
DISPLAY "The column size is ", col_info.col_size AT 2,2
DISPLAY "The column type is ", col_info.col_type AT 3,2
```

If the table "Users" was specified as:

```
CREATE TABLE users (
    userid      SERIAL,
    username    CHAR(20),
    groupid    INTEGER,
    group      CHAR(20),
)
```

Both 4GL snippets of code will result in the following display:

```
Column size is 20
Column type is 0
```



Example

This example uses database called db_examples. You can specify any other database existing on your server for this application to work. The database you refer to must already exist, this application does not create a database, it only creates tables within the specified database.

This application is important for the further progress: it must be run in order for the examples from the next chapters, which work with the same tables to function properly. If you find any problems with running the examples from the following chapters, run this example and select the "Create Tables" menu option, then exit the application without dropping the tables.

```
#####
# This example illustrates the creation and deletion of tables
# and the interconnections between the table constraints
#####

# The DATABASE statement links the source file to an existing database
# db_examples. db_examples database becomes the default and the current database
# for this program.
DATABASE db_examples
DEFINE
    info STRING,
    i INT
MAIN

# The following menu allows us to create and drop tables
# and to verify that they exist
MENU "Database"
    COMMAND "Create Tables",
        "Creates a number of tables in the database"
        CALL table_create()
    COMMAND "Drop Tables",
        "Deletes the tables from the database"
        CALL table_drop()
    COMMAND "Tables info"
        "Returns information about database tables"
        CALL table_info()
    COMMAND "Temp Table"
        "Creates a temporary table"
        LET i = i+1
        CALL table_temp()
    COMMAND "EXIT",
        "Exit the program"
        EXIT PROGRAM
END MENU
END MAIN

#####
# This function deletes the tables regardless of whether they exist
# or not
#####
FUNCTION table_drop()
    # Before we create a table, we need to delete it, if it already exists in the
```



```
# database.  
# This option is necessary, if you run this application more than once  
# and want to recreate the tables once more or just to delete them,  
# otherwise the table creation will fail as the table with the same name  
# created during the previous launch will still exist in the database.  
  
# The WHENEVER statement is required to prevent the program from termination,  
# if the tables intended for deletion below do not exist in the database  
WHENEVER ERROR CONTINUE  
    DROP TABLE country_list      -- This statement deletes the specified table,  
                                -- if it already exists  
                                -- If there is no such table, an error occurs  
                                -- which is handled by the WHENEVER statement  
                                -- above  
    DROP TABLE currency  
    DROP TABLE type_contract  
    DROP TABLE clients  
    DROP TABLE contracts  
WHENEVER ERROR STOP          -- after deleting the tables we change the  
                                -- behaviour of the program  
                                -- so now any error till cause the program to  
                                -- terminate  
  
    CALL fgl_winmessage( "Tables Dropped" ,  
                        "The tables have been dropped successfully" , "info" )  
END FUNCTION  
  
#####  
# This function creates tables, if they are absent  
# and does nothing, if they are already in the database  
#####  
FUNCTION table_create()  
  
    # We need to check whether the table exists already  
    # before creating it. If it does, the creation is skipped  
    IF fgl_find_table("country_list")= 0 THEN  
        # Table containing the names and codes of the countries  
        CREATE TABLE country_list  
        (  
            id_country  CHAR(3)  NOT NULL,   -- digital country code  
            code_lit    CHAR(3)  NOT NULL,   -- literal country code  
            country_name CHAR(50) NOT NULL,  -- country name  
  
            # creation of a unique table-level constraint for the code_lit column  
            UNIQUE (code_lit) CONSTRAINT u_country_list,  
            # creation of the table-level primary key for the id_country column  
            PRIMARY KEY (id_country) CONSTRAINT p_country_list  
        )  
        CALL fgl_winmessage( "New Table" ,  
                            "The country_list table has been created" , "info" )  
    ELSE  
        CALL fgl_winmessage( "Existing Table" ,  
                            "The country_list table already exists" , "exclamation" )  
    END IF
```



```

IF fgl_find_table("currency")= 0 THEN
    # The table of the currency codes
    CREATE TABLE currency
    (
        id_currency      CHAR(3) NOT NULL PRIMARY KEY CONSTRAINT p_currency,
                                -- digital currency code with the column-level
                                -- the primary key constraint
        id_country       CHAR(3) NOT NULL,   -- digital country code
        code_currency    CHAR(3) NOT NULL,   -- literal currency code

        # creation of the table-level foreign key for the id_country column
        FOREIGN KEY (id_country) REFERENCES country_list CONSTRAINT r_currency
    )

    CALL fgl_winmessage( "New Table",
                          "The currency table has been created", "info")
ELSE
    CALL fgl_winmessage( "Existing Table",
                          "The currency table already exists", "exclamation")
END IF

IF fgl_find_table("type_contract")= 0 THEN
    # The table containing the types of the deposit contracts
    CREATE TABLE type_contract
    (
        # creation of the primary key for the id_type column
        id_type      INTEGER NOT NULL
                    PRIMARY KEY CONSTRAINT p_type_contract, -- code of the deposit type
        name_type    CHAR(60) NOT NULL                      -- deposit type
    )

    CALL fgl_winmessage( "New Table",
                          "The type_contract table has been created", "info")
ELSE
    CALL fgl_winmessage( "Existing Table",
                          "The currency table already exists", "exclamation")
END IF

IF fgl_find_table("clients")= 0 THEN
    # The table containing the details of the clients of a virtual bank
    CREATE TABLE clients
    (
        # creation of the primary key for the id_client column
        id_client     SERIAL NOT NULL
                    PRIMARY KEY CONSTRAINT p_clients,   -- the identification number of
                                                -- the client
        id_country    CHAR(3) NOT NULL,   -- the identifier of the residence
                                                -- country
        first_name    CHAR(20) NOT NULL,  -- client's first name
        last_name     CHAR(20) NOT NULL,  -- clients last name
        date_birth    DATE,           -- birth date
        gender        CHAR(6),         -- client's sex
        id_passport   CHAR(10),       -- passport number
        date_issue   DATE,           -- date of issue of the passport
    )

```



```

date_expire  DATE,                      -- expiration date of the passport
address      CHAR(80) NOT NULL,          -- client's address
city         CHAR(20),                  -- city of residence
zip_code     CHAR(6),                  -- postal code
add_data     CHAR(512),                 -- additional data about the client
start_date   DATE        NOT NULL,      -- client's registration date
end_date     DATE        NOT NULL,      -- date of dismissal

# creation of the CHECK restriction for the gender column
CHECK (gender IN ("male", "female")) CONSTRAINT c_clients_gender,
# creation of a unique constraint for the id_passport column
UNIQUE (id_passport) CONSTRAINT u_clients_id_passp,
# Creation of the foreign key for the id_country column
FOREIGN KEY (id_country) REFERENCES country_list
    CONSTRAINT r_clients_id_count
)

CALL fgl_winmessage("New Table",
"The clients table has been created", "info")
ELSE
    CALL fgl_winmessage("Existing Table",
    "The clients table already exists", "exclamation")
END IF

IF fgl_find_table("contracts")= 0 THEN
    # The table containing the data concerning the deposit agreements of the
    # clients of a virtual bank
    CREATE TABLE contracts
    (
        id_contract  SERIAL            NOT NULL PRIMARY KEY CONSTRAINT p_contracts,
                                         -- id of the contract with the
                                         -- primary key constraint
        id_type      INTEGER           NOT NULL, -- id of the contract type
        id_client    INTEGER           NOT NULL, -- id if the client
        id_currency  CHAR(3)          NOT NULL, -- id of the currency of the
                                         -- contract
        num_con      CHAR(10)          NOT NULL, -- contract number,
        account      CHAR(14)          NOT NULL, -- contract account number,
        amount       DECIMAL(16,2)     NOT NULL, -- sum of the contract
        interest_rate DECIMAL(6,2),      -- interest rate
        start_date   DATE             NOT NULL, -- date of the contract conclusion
        end_date     DATE             NOT NULL, -- date of the contract termination

        # creation of a unique constraint for the account column
        UNIQUE (account) CONSTRAINT u_contract_account,
        # Creation of the foreign keys
        FOREIGN KEY (id_type)   REFERENCES type_contract
            CONSTRAINT r_contract_id_type,
        FOREIGN KEY (id_client) REFERENCES clients
            CONSTRAINT r_contract_id_clie,
        FOREIGN KEY (id_currency)REFERENCES currency
            CONSTRAINT r_contract_id_curr
    )
    CALL fgl_winmessage("New Table",

```



```

        "The contracts table has been created", "info" )
ELSE
    CALL fgl_winmessage( "Existing Table",
        "The contracts table already exists", "exclamation" )
END IF

CLEAR SCREEN
END FUNCTION

#####
# This function displays the information about tables and their columns
#####
FUNCTION table_info()

DEFINE
    # names of table and column selected
    tab_name, tab_column,
    # the type and the size returned
    col_type, col_size VARCHAR(25),
    type_name STRING -- used to decipher the column type

OPEN WINDOW tab_info AT 3,2 WITH FORM "table_info"
DISPLAY "!" TO info
DISPLAY "!" to qt

OPTIONS ACCEPT KEY F1

INPUT BY NAME tab_name, tab_column

# We populate the combo box with the column names
# depending on the selected table name
AFTER FIELD tab_name
CASE
    WHEN tab_name = "country_list"
        CALL fgl_list_clear("tab_column", 0)
        CALL fgl_list_insert("tab_column",1,"id_country")
        CALL fgl_list_insert("tab_column",2,"code_lit")
        CALL fgl_list_insert("tab_column",3,"country_name")
    WHEN tab_name = "currency"
        CALL fgl_list_clear("tab_column", 0)
        CALL fgl_list_insert("tab_column",1,"id_currency")
        CALL fgl_list_insert("tab_column",2,"id_country")
        CALL fgl_list_insert("tab_column",3,"code_currency")
    WHEN tab_name = "type_contract"
        CALL fgl_list_clear("tab_column", 0)
        CALL fgl_list_insert("tab_column",1,"id_type")
        CALL fgl_list_insert("tab_column",2,"name_type")
    WHEN tab_name = "clients"
        CALL fgl_list_clear("tab_column", 0)
        CALL fgl_list_insert("tab_column",1,"id_client")
        CALL fgl_list_insert("tab_column",2,"id_country")
        CALL fgl_list_insert("tab_column",3,"first_name")
        CALL fgl_list_insert("tab_column",4,"last_name")
        CALL fgl_list_insert("tab_column",5,"date_birth")
        CALL fgl_list_insert("tab_column",6,"gender")

```



```

CALL fgl_list_insert("tab_column", 7, "id_passport")
CALL fgl_list_insert("tab_column", 8, "date_issue")
CALL fgl_list_insert("tab_column", 9, "date_expire")
CALL fgl_list_insert("tab_column", 10, "address")
CALL fgl_list_insert("tab_column", 11, "city")
CALL fgl_list_insert("tab_column", 12, "zip_code")
CALL fgl_list_insert("tab_column", 13, "add_data")
CALL fgl_list_insert("tab_column", 14, "start_date")
CALL fgl_list_insert("tab_column", 15, "end_date")
WHEN tab_name = "contracts"
    CALL fgl_list_clear("tab_column", 0)
    CALL fgl_list_insert("tab_column", 1, "id_contract")
    CALL fgl_list_insert("tab_column", 2, "id_type")
    CALL fgl_list_insert("tab_column", 3, "id_client")
    CALL fgl_list_insert("tab_column", 4, "id_currency")
    CALL fgl_list_insert("tab_column", 5, "num_con")
    CALL fgl_list_insert("tab_column", 6, "account")
    CALL fgl_list_insert("tab_column", 7, "amount")
    CALL fgl_list_insert("tab_column", 8, "interest_rate")
    CALL fgl_list_insert("tab_column", 9, "start_date")
    CALL fgl_list_insert("tab_column", 10, "end_date")
END CASE

ON KEY (F10)
    EXIT INPUT

AFTER INPUT
    # First we get the information about the table column
    # selected during the input
        CALL fgl_column_info(tab_name, tab_column)
        RETURNING col_type, col_size

    # Then we need to convert the data type indexes
    # into human-readable information
        CASE col_type
            WHEN 0
                LET type_name = "CHAR"
            WHEN 1
                LET type_name = "SMALLINT"
            WHEN 2
                LET type_name = "INTEGER"
            WHEN 3
                LET type_name = "FLOAT"
            WHEN 4
                LET type_name = "SMALLFLOAT"
            WHEN 5
                LET type_name = "DECIMAL"
            WHEN 6
                LET type_name = "SERIAL"
            WHEN 7
                LET type_name = "DATE"
            WHEN 8
                LET type_name = "MONEY"
            WHEN 10

```



```

        LET type_name = "DATETIME"
WHEN 11
        LET type_name = "BYTE"
WHEN 12
        LET type_name = "TEXT"
WHEN 13
        LET type_name = "VARCHAR"
WHEN 14
        LET type_name = "INTERVAL"
WHEN 15
        LET type_name = "NCHAR"
WHEN 16
        LET type_name = "NVARCHAR"
WHEN 45
        LET type_name = "BOOLEAN"
# In case the user selects this option before
# creating tables
WHEN -1
        LET tab_name = "<table not found>"
        LET tab_column = "<column not found>"
        LET type_name = "N/A"
        LET col_size = "N/A"
END CASE

# And then we display the information about the selected column
LET info = "The data for column " || tab_column ||
            "\n of table " || tab_name || "\n are as follows: " ||
            "\n Column type: " || type_name ||
            "\n Column size: " || col_size
CALL fgl_winmessage("Column Info", info, "info")

CONTINUE INPUT -- we loop the input to allow the user
                -- to get information about other columns
END INPUT

CLOSE WINDOW tab_info
CLEAR SCREEN

END FUNCTION

#####
# This function creates a temporary table which need not be dropped
#####
FUNCTION table_temp()

# We ensure that the table is created only once
# and do not drop this table explicitly
# We cannot use the fgl_table_info() function
# for a temporary table
IF i <= 1 THEN
    # The temporary table is deleted,
    # when the program is terminated
    CREATE TEMP TABLE my_temp_table(
        id_col SERIAL,

```



```
    info_col CHAR(20),
    num_col INT
)
CALL fgl_winmessage( "Temporary Table",
    "The temporary table has been created", "info")
ELSE
    LET info = "The temporary table already exists.",
        "\n If you restart the program, it will be deleted",
        "\n then you will be able to create it again."
    CALL fgl_winmessage( "Temporary Table", info, "exclamation")
END IF

END FUNCTION
```

The Form File

The form file is called "table_info.per" and is used to select the table and the column which parameters are requested.

```
DATABASE formonly

SCREEN {
    Table name: [tab_name]
    Table column: [tab_column]

    [info] [qt]
}

ATTRIBUTES

# We include all our tables into the first combo box widget
tab_name = formonly.tab_name, widget = "combo", REQUIRED,
           include = ("country_list", "currency", "type_contract", "clients",
"contracts");
# and fill the second combo box widget depending on the first choice
tab_column = formonly.tab_column, REQUIRED, widget = "combo", include =
("empty");
info = formonly.info, widget = "button", config = "F1 {Get Info}";
qt = formonly.qt, widget = "button", config = "F10 {QUIT}";
```



Populating Database Tables

An ability to modify the information stored in a database is very important for the successful database development and usage. 4GL provides a set of tools that are used to populate database tables with values and to retrieve these values. In this chapter we will discuss some statements and options that can be used in these purposes.

The LOAD Statement

It is obvious, that a database must include some data. These data can be entered by the user during the program runtime or loaded from some external sources. The LOAD statement is used to fill the existing table with the data stored in the specified file.

The syntax of the LOAD statement is as follows:

```
LOAD FROM filename [DELIMITER clause] INSERT INTO table_name [(columns)]
```

The *filename* here stands for a quoted character string or a variable of CHAR or VARCHAR data type, which specifies the name and the extension of the file that contains the data to be loaded. If the input file and the 4GL file are located in different folders, the *filename* must include the path.

The *table name* stands for the name of a table into which you want to load the data from the file, and the *columns* is the list of columns separated by commas into which the data will be inserted. The column list may be omitted, if you want to insert the data in all the columns of the specified table.

The INSERT clause can be replaced by a variable of character data type which contains the text of the INSERT clause. It is used to specify the place in the database where the loaded data will be stored. The LOAD statement cannot overwrite the existing rows; it creates new ones within the specified table. Thus, if you load the same file twice, the table will contain the rows with duplicating values. If any column has a UNIQUE or PRIMARY KEY constraint, you will not be able to load the same file for the second time and will get a runtime error instead.

The Loaded File

The *filename* that follows the LOAD FROM keywords must indicate a name of the file which contains ASCII characters or characters, supported by the locale. These characters represent the values that are to be inserted to the table, so they are to be of the same or of compatible data types with those of the receiving columns. The table below describes the main rules of data representation in an input file:

Data Type	Input Format
CHAR, VARCHAR, TEXT	Values intended to be inserted in CHAR, VARCHAR, and TEXT columns can contain any printable symbols, but you have to put a backslash (\) before any backslash or delimiter symbol you want to be displayed. You should also place the backslash at the very end of variables that belong to the TEXT value. You can create a blank value by putting any number of blank spaces between two delimiters, but you shouldn't put blank spaces at the very beginning of



	CHAR, VARCHAR, or TEXT values.
	The values of these data types may include more characters than the column maximum length supports, but they will be truncated from right when loaded to the table.
DATE	By default, these values should be specified in the following format: <i>mm/dd/yyyy</i> or <i>mm/dd/yy</i> . This is the default format for the English locale. The other locales may support other default date formats.
	The value specified here must be a realistic date. Such dates as May 32 won't be accepted by the database.
DATETIME, INTERVAL	The required format for DATETIME variables is <i>YYYY-MM-DD hh:mm:ss.ppp</i> . The required format for the INTERVAL variables is <i>YYYY-MM</i> or <i>dd hh:mm:ss.ppp</i> . You can also use a contiguous subset of date and time units to compose a DATETIME or INTERVAL value.
	If the input file specifies some units that are outside the declared precision of the column, they will be ignored.
MONEY	The values that specify amounts of money may be preceded with currency symbols, but they are not necessary.
SERIAL	If you want the database server to supply a new SERIAL number, put 0 as the SERIAL value. You can set a SERIAL number to a positive literal integer, but an error will occur, if this number matches an existing SERIAL value.
BYTE	The values of the BYTE data type shouldn't be preceded or followed by blank spaces and must be represented by ASCII-hexadecimals.

The extension of the input file should be *.unl*. Create a text document which contains all the necessary data and save the document with the extension *.unl*. The values stored in the file should be separated with delimiters that match the delimiters specified in the LOAD statement. If no DELIMITER clause is specified, the program takes the pipe symbol (|) as the default delimiter.

Each sequence of values which represents a separate row is called *an input record*. Each new record should be placed on a separate line or be separated from other records with the NEWLINE character (ASCII-10). Each value of a record must be followed by a delimiter including the last value of the record. If you want a field of the table row to remain empty, put two delimiters one by one and do not put anything between them.

The content of an input file may look as follows:

```
Ann|Smith|21|
||Bell|30|
Kate|Marson||
Dominic| 45|
```

This file contains a list of values that can be assigned to four rows of a three-column database table:

- The first row specifies values for the whole row.
- The second row specifies values for the second and the third column
- The third row specifies values for the first and the second column
- The fourth row specifies values for the first and the third column



Below, you can see an example of the simplest statement that will insert these values to the database table *my_table*:

```
LOAD FROM "load_file.unl" INSERT INTO my_table
```

If the input file and the executable source file are located in different folders on your computer, you should specify the path to the file:

```
LOAD FROM "C:/WORK/workspace/DataBases/text/load_file.unl" DELIMITER "|"  
INSERT INTO my_table
```

The DELIMITER Clause

The DELIMITER clause is optional and has the following syntax:

```
DELIMITER delimiter_symbol
```

The delimiter symbol stands for a quoted string or a variable of CHAR or VARCHAR data type, which contains the delimiter symbol. Unlike the DELIMITERS clause of the INSTRUCTIONS section of the form specification file, the quoted string here doesn't need to consist of two symbols:

```
DEFINE del CHAR  
LET del = "^"  
  
LOAD FROM "file1.unl" DELIMITER ","...  
LOAD FROM "file2.unl" DELIMITER del...
```

The delimiter symbol cannot be represented by hexadecimal numbers (A through F, a through f, 0 through 9), NEWLINE symbol, Backslash, or NEWLINE character.

Remember, that the delimiters specified in the LOAD statement and used in the input file must match. Thus, if you specify "^" as the delimiter, the loaded file must use this symbol as the delimiter:

```
LOAD FROM "file.unl" DELIMITER "^"...
```

```
file.unl:  
Ann^Smith^21^  
^Bell^30^  
Kate^Marson^^  
Dominic^^45^
```

Remember, that the backslash is used as a symbol that makes the program treat the following symbol as literal. If a value in an input file contains a value which has a delimiter or NEWLINE symbol without a preceding backslash, an error occurs.

The INSERT Clause

The INSERT clause is used to specify the table and columns in which the new data are to be stored. If the number of values in one input record matches the number of columns in the table, the INSERT clause requires only the table specification:

```
LOAD FROM "general.unl" DELIMITER "^" INSERT INTO address_table
```



As it has been said at the beginning of this chapter, the INSERT clause can be replaced by a CHAR or VARCHAR variable which contains such clause. The value of this variable should be an exact copy of the source code it is intended to replace:

```
DEFINE ins VARCHAR(20)
LET ins = "INSERT INTO address_table" -- assigning an INSERT clause
-- specification to a variable
...
...
LOAD FROM "general.unl" DELIMITER "^" ins -- referring to the INSERT clause
-- by means of a variable
```

Sometimes, it is necessary to specify column names in the INSERT clause. You have to list the column names in the parentheses following the table name, if:

- You want to insert data to some, but not to all of the column tables;
- The order of the data stored in the input file doesn't correspond to the order of the columns in the table. In such case you need to list the columns in the correct order.

In the following example, the program is directed to insert values only to the columns *country*, *city*, *zipcode*:

```
LOAD FROM "load_file.unl" DELIMITER ":"  
INSERT INTO address_table(country, city, zipcode)
```

Introduction into the SELECT Statement

The SELECT statement is the primary tool of referring and querying the information stored in a database. It is one of the SQL statements used by 4GL and can be fairly called the most important one. SELECT is also one of the most complex of the SQL statements, and we will discuss the statement, its application and clauses in several chapters.

The general syntax of the simplest SELECT statement used to retrieve a single row from a database table is:

```
SELECT select_list INTO clause FROM clause WHERE clause
```

A detached SELECT statement is used to reference a table and retrieve a row of values from it typically has three clauses: INTO clause, FROM clause and WHERE clause. Though the INTO clause is regarded as an optional one in some cases discussed further, in this case it is obligatory.

	Note: Remember, that the SELECT statement cannot change the information stored in the database. It can only query the data.
---	--

The SELECT statement which we discuss in this chapter can be used to select only one row from the table specified in the FROM clause. To do that, we need the WHERE clause which contains the condition specifying which row to select. This clause is described later in this chapter. If the table contains several rows that meet the specified conditions, a compile-time error will occur.



The Select List

The select list is used to determine the set of database objects from which the information is to be selected by the statement. It can contain one or more column names, preceded with table names, if the column names are not unique within the current database. It can also contain the asterisk (*) which indicates that the statement has to select all the columns that comprise the specified table, or be represented by an arithmetic expression:

```
SELECT f_name          -- select values from a specified column
SELECT f_name, l_name  -- select values from several columns
SELECT *               -- select values from all the columns of
the table
SELECT custom.f_name   -- a column name is preceded by a table name
SELECT ordz.price*0.75 -- an arithmetic expression with a column
```

The order in which the SELECT statement queries the columns is determined by the order in which they are listed in the SELECT clause. This order can correspond to the initial columns order in a table or can be changed by swapping the column names in the select list. For example, if the table was specified as:

```
CREATE TABLE custom (
    id INT;
    f_name CHAR(20);
    l_name CHAR(20);
    age INT
)
```

The select list can be represented by an asterisk (*) or by the list of all the columns (i.e. SELECT id, f_name, l_name, age...). In this case the columns will be queried in the same order in which they are present in the table.

You can change the order in which the query is performed, i.e.:

```
SELECT l_name, f_name, id...
```

The program will take values from the columns in the order which is specified in the statement.

The INTO Clause

The INTO clause contains a list of program variables which will store the data selected from the columns specified in the select list. Here is the simplest structure of the INTO clause of the SELECT statement:

```
SELECT ... INTO variable_list
```

The variable list contains identifiers of one or more program variables which will receive the values of the columns, specified in the SELECT clause. The order, number, and data types of the variables given in the variable list should match those of the columns specified in the select list.

We will illustrate the work of the INTO clause of the SELECT statement by querying the table with the following structure:

```
CREATE TABLE custom (
```



```
id INT;
f_name CHAR(20);
l_name CHAR(20);
age INT
)
```

The following SELECT...INTO... clauses are valid for the table given above:

```
DEFINE u_id, u_age INT,
      fname, lname CHAR (20)
      ...
      ...

SELECT * INTO u_id, fname, lname, u_age FROM custom
SELECT f_name, l_name INTO fname, lname FROM custom
SELECT age, l_name INTO age, l_name FROM custom
```



Note: The SELECT statement cannot include only the SELECT and INCLUDE clauses. It needs at least one more clause for the successful operation. In this case the FROM clause is required. It will be described in details in the following section.

If the variable list contains a program record or a program array identifier, the number, order, and data type of array elements or array records must match the number of corresponding fields:

```
DEFINE pers RECORD
      fname, lname CHAR (20)
      END RECORD

      ...
      ...

SELECT f_name, l_name INTO pers FROM custom
```

You can also specify a member of the record or an element of an array as an item of the variable list:

```
SELECT l_name INTO pers_arr[2] FROM custom
```

The FROM Clause

The FROM clause contains the names of the tables where the column names from the select list are situated. The FROM clause has the following structure:

```
SELECT...FROM table_name [,table_name...]
```

Now, when you have a notion of all the necessary clauses of the SELECT statement, you can see an example of the valid SELECT statement specification:

```
DEFINE fname, lname CHAR(20)
...
SELECT f_name, l_name INTO fname, lname FROM staff
DISPLAY "NAME: ", f_name, " Last name: ", l_name AT 2,2
```



This statement queries the columns *f_name* and *l_name* located in the table *staff* and assigns the values of the first rows of these columns to program variables *fname* and *lname*, correspondingly. The DISPLAY statement will display the retrieved values to the screen.

You can display the values retrieved from a database table to a screen form. To display values to a screen record, use the DISPLAY TO statement and a list of variables you want to display. If you want to display values to a screen array, they must be stored to program array and then displayed by means of the DISPLAY ARRAY statement. Here is an example of how it can be done:

```
DEFINE pers ARRAY [5] OF RECORD
    id INT,
    fname, lname CHAR (20),
    age INT
END RECORD,
test1_c, test1_r CHAR(20)

SELECT * INTO pers FROM custom

OPEN WINDOW my_w WITH FORM "db_test"
CALL set_count(5)
DISPLAY ARRAY pers TO scr_rec.*
```

This SELECT statement copies only the first row of the database table *custom*, so only the first element of the array will get values. The result of the DISPLAY ARRAY statement execution can look as follows:

Personal			
ID	F_Name	L_Name	AGE
1	Alisa	Miller	15
0			0
0			0
0			0
0			0



Note: Keep in mind, that the number, the data types, and the order of array elements or record members should correspond to the number, data types and order of table columns. Otherwise, a run-time error will occur.

The WHERE Clause

The WHERE clause is used to specify the criteria for the program to follow when executing the SELECT statement. Whereas this clause is optional for some cases of the SELECT clauses of other statements, a stand-alone SELECT statement used without a cursor (which is discussed later in this manual) and intended to retrieve exactly one row must have it. In this section, we will discuss the conditions under which a single row will be selected from the table.

The WHERE clause can include one or several comparisons and/or conditions. When a program executes the SELECT statement, it chooses the rows which values satisfy the given conditions.

The conditions that can be included into a WHEN clause are as follows:



- *The condition including a relational operator.* Such condition is satisfied when the statement finds a row, for which the Boolean expression following the WHERE keyword is TRUE:

```
SELECT * INTO pers FROM custom WHERE id >4
```

The statement will get the first row from the *custom* table, with the *id* column value greater than 4, which will be "5":

ID	F_Name	L_Name	AGE
5	Mark	Orio	23
0			0
8			8

- *Range condition.* This condition is specified using the [BETWEEN...AND](#) operator. It is satisfied when the program finds a row containing a value that is within a specified range:

```
SELECT * INTO pers FROM custom WHERE id BETWEEN 2 AND 4
```

- *Membership condition.* This condition is specified using the [IN\(\)](#) operator. It is considered to be satisfied when the expression to the left from the IN keyword matches or is equal to one of the values specified to the right from the IN keyword:

```
SELECT * INTO pers FROM custom WHERE f_name
      IN (John, Marko, Kate, Josephine)
```

- *IS NULL condition.* The IS NULL condition is satisfied when the column specified to the left from the IS NULL keywords contains a null value. You can use the NOT keyword before the IS NULL keywords in order to select a row which does not contain a NULL value in the specified column:

```
SELECT * INTO pers FROM custom WHERE age IS NULL
SELECT * INTO pers FROM custom WHERE age IS NOT NULL
```

- *LIKE or MATCHES condition.* These conditions are considered to be satisfied when a character value in a column specified to the left from the [LIKE or MATCHES](#) operator matches the quoted string or a value of the character-type variable specified to the right from the keywords:

```
SELECT * INTO pers FROM custom WHERE l_name MATCHES "O*"
```

This SELECT statement will chose the row which has the value beginning with the letter *O* in the column *_name*.

The rules and restrictions for the LIKE and MATCHES keywords usage are similar to those described in the "LIKE and MATCHES operators" section of Chapter 10.

The right-hand operands of the conditions listed above can be represented not only by values or variables, but also by the column names:

```
SELECT * INTO fam FROM fam_tree WHERE fath_name MATCHES child_name
```



When executing this statement, the program will compare the value of the column *father_name* to the value of the column *child_name* and select the first row where the values in these columns match.

You can also use logical operators, such as AND, OR, and NOT within the WHERE clause in order to combine several conditions and specify more precise criteria of the rows choice:

```
SELECT * INTO pers FROM staff WHERE age>30 AND nam MATCHES "A*"
```

Additional SELECT Facilities

There are a number of features supported by the SELECT statement that can influence the results that are passed to the variables.

Selecting Column Substrings

You can select and pass only a part of a value stored in a column which supports the CHARACTER data type. For example, if you want to have an abbreviation of a customer's city, but not the whole name, you can specify the substring of a necessary length in square brackets following the column name:

```
SELECT city[1,3] INTO cus_cit FROM custom
```

If the value in the column *city* is "London", the variable *cus_cit* gets the value "Lon"

Changing the Selected Values

You can change the values taken from the database table before they are sent to the variables. You can use arithmetic operators and built-in functions in order to do it.

When you want to change the numeric value of the column, specify the necessary arithmetic operation after the column name in the SELECT statement.

For example, the table contains the column *payout* where the values are given in Euros, and you want the resulting value to be given in US dollars. If the Euro-to-dollar rate is 1:1.2, your query can be given as follows:

```
SELECT pay_eur*1.2 INTO pay_dol FROM finance_report
```

If the first value in the column *pay_eur* is 300, the value assigned to the variable *pay_dol* will be 360.

The items listed in the SELECT clause can also be represented as operands of different built-in functions. For example, you can make the program select only the YEAR part of a date column, if the day and the month are not important for you:

```
SELECT year(age) INTO pers.age FROM custom
```

Querying Several Tables

The ability to query database for values would be insufficient without an ability to refer to several tables. You can use the SELECT statement to create a query which will retrieve data from different tables. To do this, list the columns you want to chose in the SELECT clause and the tables you want to query - in the FROM clause. The order of the columns should match the order of the corresponding tables:

```
DEFINE rec1 RECORD num1 int, v1, v2 CHAR(10) END RECORD,
```



```
rec2 RECORD num2 int, v3, v4, v5 CHAR(10) END RECORD
```

```
SELECT * INTO rec1, rec2 FROM table1, table2
DISPLAY rec1.*  
DISPLAY rec2.*
```

The snippet of the source code given above queries *table1* and *table2* and sends the values of the first rows of all their columns to program records *rec1* and *rec2*. It is important, that the number of values in each record should match the number of columns in the corresponding table.

The following example demonstrates how you can make the SELECT statement query specified columns of different tables:

```
SELECT lnam, work INTO nam, empl FROM personal, jobs
```

When the program starts executing this statement, it will take a value from the column *name* in a table *personal* and assign it to the variable *lnam*; the value of the column *work* in the table *jobs* will be assigned to the variable *empl*.

Selecting into a Temporary Table

The SELECT statement can have the INTO TEMP clause which should follow any other optional clauses of the SELECT statement. This clause instructs the program to store the results of the query in a temporary table. This method is similar to the SELECT statement with the INTO clause which specifies the name of another table. However, in the case of INTO TEMP clause, the table containing the results of the query is deleted when the program is terminated. The syntax is as follows:

```
SELECT select_list FROM table_name [WHERE clause] INTO TEMP temp_table
```

The INTO TEMP clause creates a temporary table with "temp_table" name that contains the query results. This temporary table is similar to temporary tables created using the CREATE TEMP TABLE statement. The columns in this temporary table have the same name as the columns from the select list.

If you use the same query results more than once, using a temporary table saves time. In addition, using an INTO TEMP clause often gives you clearer and more understandable SELECT statements. However, the data in the temporary table is static; data is not updated as changes are made to the tables used to build the temporary table.



Note: You must not use the INTO and INTO TEMP clause in one and the same SELECT statement, because this will cause an error.

The example below will create temporary table *my_temp* with 3 columns: *column_1*, *column_2*, *column_3*. It is important that *sel_tab* should also contain the columns with such names.

```
SELECT column_1, column_2, column_3
FROM my_tab INTO TEMP my_temp
```



A Table Alias

In some cases you may need to use a table alias in the SELECT statement instead of the table name. This will typically happen when a column in a table has the same name as one of the variables declared in the program. In the following example the SELECT statement will search for the row where the `client_id` is 5 and not 3:

```
DEFINE client_id
LET client_id = 5
SELECT * INTO rec.* FROM clients WHERE client_id=3
```

In such cases we need to introduce the table alias which makes the code more accurate. We add this alias before each item in the select list in the way a table name is added to an identifier; in our case it will look like this:

```
SELECT alias.*
```

We also need to declare it in the FROM clause, the alias name should follow the table name for which it is created, e.g.:

```
FROM clients alias1, items alias2
```

Then we can use this alias to specify the WHERE conditions. The complete SELECT statement with an alias will look as follows:

```
SELECT alias.* INTO rec.* FROM clients alias WHERE
alias.client_id=3
```

The UNLOAD Statement

It is often necessary to send the data stored in a database to a file that can be easily opened, studied, and corrected. The UNLOAD statement can be used in these purposes: it sends the data of the current database to the specified file. In this way you are able to see the complete contents of a table.

The general syntax of the UNLOAD statement is:

```
UNLOAD TO filename [DELIMITER delimiter] SELECT clause
```

The *filename* is represented by a quoted string which specifies the name and the extension of the file to where the data should be copied. The *filename* can also be represented by a variable of CHAR or VARCHAR data type the value of which contains the name and the extension of this file. The character string or the variable value can include a path, if necessary.

The SELECT clause specifies the rows that are to be copied to the output file. It is necessary for the UNLOAD statement operation.

Two conditions must be met before the program copies the data from a database to a file. The first condition is that the database should be activated by means of the DATABASE statement before the UNLOAD statement execution. The second condition is that the user must have SELECT privileges for each column that is specified in the SELECT clause of the UNLOAD statement.

After the program executes the UNLOAD statement, it doesn't delete the copied data from the table.



The Output File

The file which is purposed to store data copied from the database, is specified by the *filename*. In the U.S. English locale, the values stored in this file consist of ASCII characters. In other locales, they can include characters from the locale code set.

A set of output values which represent a row from the database table is called an *output record*. Each output record is terminated by a newline character (which is ASCII 10).

The values of different data types are represented in the output file according to the following rules:

Data Type	Output Format
CHARACTER, VARCHAR,	If the values of CHAR or TEXT (but not VARCHAR) data type end with blank spaces, they are truncated. If a value contains a delimiter character or a literal backslash (\), the program adds another backslash before it. A backslash is also added at the end of the TEXT value and before a newline character of the VARCHAR value.
TEXT	
INTEGER, SMALLINT, DECIMAL, FLOAT, SMALLFLOAT, MONEY	The values of these data types are saved as literals, no blank spaces preceding them. No currency symbol is added before the MONEY value. If an INTEGER or SMALLINT value has zero value, it is represented as 0. Zero values for DECIMAL, FLOAT, SMALLFLOAT, and MONEY are represented as 0.00.
DATE	SERIAL values, if any, are transformed into literal integers when saved to an output file. The values of the DATE data type are represented as <i>mm/dd/yyyy</i> unless you specify another format.
DATETIME, INTERVAL	The values of the DATETIME data type are sent to the output file in the format <i>yyyy-mm-dd hh:mm:ss.ppp</i> or as a contiguous subset of given time units.
BYTE	INTERVAL values will have the format <i>yyyy-mm</i> or <i>dd hh:mm:ss.ppp</i> or will be presented as a contiguous subset of possible time units. If a table column stores BYTE values, they are sent to the output file as ASCII hexadecimal characters. The logical record of BYTE values can quite long and inconvenient to edit or to print.

If any table row contains null values, they will be represented as two consecutive delimiters; no character or whitespace will be added between them.

The SELECT Clause

Each UNLOAD statement must contain the SELECT clause which syntax is as follows:

```
SELECT select_list FROM table_name [WHERE clause]
```

You can see that the SELECT clause does not include the INTO clause. It is absent because the output file serves as the container for the selected values. The rest of the rules introduced for a stand-alone SELECT statement are also true for the SELECT clause of the UNLOAD statement except that the FROM clause can contain only one table name.

The SELECT clause contains the select list which specifies from which columns the values should be taken and the FROM clause specifying the table name. Unlike a stand-alone SELECT statement, the SELECT clause of the UNLOAD statement can retrieve more than one row. It retrieves all the rows from the specified table and writes them to the output file, if no condition is specified. If the WHERE clause is present, it retrieves all the rows meeting the condition and writes them to the file.



When you specify a filename in the UNLOAD statement, you should quote it and specify its extension. The following snippet of the source code illustrates how the program copies the rows in which the value in the column "city" is "NY":

```
UNLOAD TO "pers_file.unl" DELIMITER ","
SELECT * FROM pers_dat WHERE city = "NY"
```

These lines must produce an output file which can have the following content:

```
Jackson and Co,private enterprise,NY,1965,
Lilly,Ltd,NY,2003,
Richards and Sons,private enterprise,NY,1987,
```

The DELIMITER Clause

The DELIMITER clause is optional and has the following syntax:

```
DELIMITER delimiter_symbol
```

The *delimiter_symbol* is represented by a quoted literal symbol or by CHAR or VARCHAR variable which contains such symbol. The program will use the specified delimiter to separate the values of different columns within one row in the output file. The requirements on UNLOAD delimiters are similar to those of the LOAD statement. You should include this clause, if you want the values in the output file to be separated by any delimiter other than the default one which is the pipe symbol (|).

Host Variables

There are cases when it is necessary to make the SELECT statement dynamic, to let the user or the program influence the result of the statement execution. If you want to do it, you can use *host variables* - the variables that send their values to the statement when it is executed. Usually these host variables are applied to the WHERE clause of the SELECT statement or the SELECT clause of the UPLOAD statement to make the condition dynamic.

The host variables can get their values from both the program and user input. Here is an example of host variables usage within the UNLOAD statement:

```
DEFINE      city char(20),
            c integer

LET city = "DC"
PROMPT "How old should the claimant be?" for c

UNLOAD TO "unload_file.unl" DELIMITER ","
SELECT * FROM my_first WHERE age = c and city = a
```

The program will choose the rows where the values in the column *city* are "DC" and in the column "age" match the number entered by the user.

SQL INTERRUPT Option

The execution of SQL statements can be interrupted by the user as well as the 4GL statements.



This ability can be enabled or disabled by the SQL INTERRUPT ON/OFF clauses of the OPTION statement which specify whether the Interrupt key (CONTROL-\) can be used to interrupt SQL statements. The default setting for the SQL INTERRUPT option is OFF, but it can be changed in the following way:

```
MAIN
DEFINE...
OPTIONS SQL INTERRUPT ON
...
END MAIN
```

If the SQL interruption is enabled and the user presses the Interrupt key while an SQL statement execution (i.e. while the SELECT statement is performing a query), the program waits until the database server completes the SQL statement and then executes the interruption.

If the [DEFER INTERRUPT](#) statement isn't executed before the user presses the Interrupt key, 4GL terminates the program execution. If the DEFER INTERRUPT statement was activated, 4GL assigns the value TRUE to the built-in variable *int_flag* and the program execution isn't terminated.

It can be useful to enable the SQL INTERRUPT option only for a certain part of the program, so you can disable it by another OPTIONS statement:

```
OPTIONS SQL INTERRUPT ON
...
...
...
OPTIONS SQL INTERRUPT OFF
```

Example

This example is a logical continuation to the example in the previous chapter. It populates the tables created by the previous example and thus you need to run the previous example on the same database before running the example below.

	Note: The database db_examples must exist on your database server. It can be any other existing database as long as you change the DATABASE statement to refer to it instead of the db_examples.
---	---

If something went wrong, you should relaunch the application from the previous chapter and activate the "Drop Tables" option followed by the "Create Tables" option to recreate the tables.

This application is important for executing the example applications from the following chapters. Together with the application from the "Databases in 4GL" chapter it constitutes the core

```
#####
# This application loads the data into tables and then unloads them to other
# files. It is recommended that you run this application in GUI mode.
# Note: To be able to execute this program, you should run the application from
# the previous chapter first!
#####
DATABASE db_examples
```

```
MAIN
DEFINE
```



```
r_clients RECORD -- the structure of the record is the same as the
-- structure of clients table
    id_client      INTEGER,
    id_country     CHAR(3),
    first_name     CHAR(20),
    last_name      CHAR(20),
    date_birth     DATE,
    gender         CHAR(6),
    id_passport    CHAR(10),
    date_issue     DATE,
    date_expire    DATE,
    address        CHAR(80),
    city           CHAR(20),
    zip_code       CHAR(6),
    add_data       CHAR(512),
    start_date     DATE,
    end_date       DATE
END RECORD,
v_country_name   CHAR(50),
code_currency    CHAR(3),
num_key          INT,
id_entered      SMALLINT

IF fgl_fglgui() = 0 THEN
    CALL fgl_winmessage("Character Mode",
        "It is recommended that you run this application in GUI mode" ||
        "\nbecause most of the information is displayed to the console",
        "info")
END IF

MENU "Populating"
COMMAND "Load Records"
    "Loads the records from the unl files into tables"
    # We use the LOAD statement, to load the information from the
    # previously created .unl files.
    # The structure of each .unl file must correspond to the structure of
    # the table.
    LOAD FROM "country_list.unl"    INSERT INTO country_list
    # The process of loading is reflected in console
    DISPLAY "country_list loaded"
    LOAD FROM "currency.unl"        INSERT INTO currency
    DISPLAY "currency loaded"
    LOAD FROM "type_contract.unl"   INSERT INTO type_contract
    DISPLAY "type_contract loaded"
    LOAD FROM "clients.unl"        INSERT INTO clients
    DISPLAY "clients loaded"
    LOAD FROM "contracts.unl"      INSERT INTO contracts
    DISPLAY "contracts loaded"

COMMAND "Unload Records",
    "Records the table contents into files"
    # If the files specified in the UNLOAD statement were not created
    # beforehand, they would be created automatically.
    # If you run the program in character mode, they will be created
    # in the output folder of your project, if you run the program
```



```
# in the GUI mode, they are created on the application server.  
# Most likely they will be created in the following  
# directory: C:\Users\All Users\Querix\Lycia\progs  
  
UNLOAD TO "country_list_new.unl"    SELECT * FROM country_list  
UNLOAD TO "currency_new.unl"        SELECT * FROM currency  
UNLOAD TO "type_contract_new.unl"  SELECT * FROM type_contract  
UNLOAD TO "clients_new.unl"        SELECT * FROM clients  
UNLOAD TO "contracts_new.unl"      SELECT * FROM contracts  
  
CALL fgl_winmessage("Unloading",  
                    "The files with the tables contents were created." ||  
                    "\n They are located in your project output folder" ||  
                    "\n or on your application server, depending on the mode.",  
                    "info")  
DISPLAY fgl_file_dialog("open", 1, "UNL Files",  
                        "C:\Users\All Users\Querix\Lycia\progs", "clients_new.unl",  
                        "unl files (*.unl||")  
  
COMMAND "Delete Records",  
        "Deletes all the records from all the tables"  
# Before we load the data into tables we delete all the entries the  
# tables might have.  
# This precaution is needed, if you run this application more than  
# once. The DELETE statement proper will be discussed later  
DELETE FROM contracts  
# The process of deleting is reflected in console  
DISPLAY "Deleted from contracts"  
DELETE FROM clients  
DISPLAY "Deleted from clients"  
DELETE FROM currency  
DISPLAY "Deleted from currency"  
DELETE FROM country_list  
DISPLAY "Deleted from country_list"  
DELETE FROM type_contract  
DISPLAY "Deleted from type_contract"  
  
COMMAND "SELECT Queries",  
        "Examples of the SELECT queries"  
MENU "SELECT"  
# Note: if you are getting an empty record or a record  
# with inadequate data, recreate the tables with the previous  
# example application and reload them with the values using  
# the current application  
BEFORE MENU  
# One of the options works only in character mode due  
# to some peculiarities, so it will be hidden from the  
# user if the application is run in GUI mode  
IF fgl_fglgui()= 1 THEN  
    HIDE OPTION "Looped SELECT"  
END IF  
  
COMMAND "Selecting a Record",  
        "Selects a complete record from a table"  
LABEL id_repeat:
```



```
# Here you can enter the search criteria for the
# WHERE clause of the SELECT statement
LET id_entered=fgl_winprompt(5,8,
    "Enter the client ID number between 1 and 10", "1", 3, 1)

IF id_entered <1 OR id_entered > 10 THEN
    CALL fgl_winnmessage("Wrong ID",
        "You should enter ID from 1 to 10", "exclamation")
    GOTO id_repeat
END IF

SELECT c.* -- here we use "c" as the alias for the
-- table and use * to retrieve all the values
INTO r_clients.* -- we retrieve the values into the record
FROM clients c -- here we declare the alias
WHERE c.id_client = id_entered
-- and use it before the column name
-- instead of the table name

DISPLAY "SELECT is used to retrieve one record ",
    "from clients table"
DISPLAY "according to the entered ID:"
DISPLAY "SELECT c.* INTO r_clients.* FROM clients c",
    "WHERE c.id_client = ", id_entered
DISPLAY "Fetched row:"

DISPLAY r_clients.*
INITIALIZE r_clients.* TO NULL
-- reset the record to get it ready
-- for another query

COMMAND "Selecting One Value",
    "Selects a single value from a specific record"

LABEL country_repeat:
# Here you can enter the search criteria for the
# WHERE clause of the SELECT statement
LET id_entered=fgl_winprompt(5,8,
    "Enter the country ID number", "840", 3, 1)

# We select a value of a single column
SELECT country_name
    INTO v_country_name -- into a single variable
    FROM country_list
    WHERE id_country = id_entered

IF v_country_name IS NULL THEN
    CALL fgl_winnmessage("Wrong ID",
        "You should enter a valid ID", "exclamation")
    GOTO country_repeat
END IF

# We may add the name of the table before the
# column name here, but it is not necessary
```



```
SELECT country_list.country_name
      -- we select a value of a single column
      INTO v_country_name    -- into a single variable
      FROM country_list
      WHERE id_country = "840"

DISPLAY "SELECT is used to retrieve one record ",
        "from clients table"
DISPLAY "according to the entered ID:"
DISPLAY "SELECT country_name INTO v_country_name ",
        "FROM country_list WHERE id_country = ",
        id_entered
DISPLAY "The fetched value:   ", v_country_name

INITIALIZE v_country_name TO NULL
      -- reset the value to get it
      -- ready for another query

# This option is hidden in the GUI mode
# using the BEFORE MENU clause
COMMAND "Looped SELECT",
        "The SELECT statement will be executed in a loop"
DEFER INTERRUPT    -- we disable normal operation of the
                    -- Interrupt key
OPTIONS
SQL INTERRUPT ON -- Then we enable the SQL INTERRUPT
                  -- which will cause the
                  -- Interrupt key to interrupt only SQL
                  -- statements. The program will
                  -- continue after the currently
                  -- executed SQL statement is interrupted.

CALL fgl_winmessage("Interrupting",
                    "Press CONTROL-C to interrupt the loop", "info" )
WHILE TRUE
    SELECT c.* INTO r_clients.* FROM clients c
    WHERE c.id_client = 1

    DISPLAY r_clients.id_client, " ",
            r_clients.first_name CLIPPED, " ",
            r_clients.last_name CLIPPED, " ",
            r_clients.address
    IF int_flag <> 0 -- we specify that the loop should
                      -- be terminated, if the
                      -- Interrupt key is pressed.
        THEN EXIT WHILE
    END IF
END WHILE

LET int_flag = 0 -- we reset the flag to 0 to be able
                 -- to reuse the Interrupt key

COMMAND "Return",
        "Return to the previous menu"
EXIT MENU
```



```
END MENU

COMMAND "EXIT"
    EXIT PROGRAM

END MENU

END MAIN
```

UNL Files

Here are the files from which the data are loaded into the tables and their contents. You need to have these files in your project source folder, make sure that they are in the same folder as the source file, or edit the code to add the path to the files.

Remember, that one line in the load file should correspond to one row in the table. In order to avoid mistakes, just copy the contents given below to your load files using copy and paste keys. If you want to add the contents to the load files manually, be sure that each line corresponds to one row.

clients.unl:

```
0|840|Jimmy|Smith|09/09/1950|male|123456|01/01/1988|01/01/2018|Village Rd. 202,
app.12|New Work|13786|N/A|09/10/2010|12/31/9999|
0|826|Katherin|Stevens|07/20/1979|female|234567|08/02/1997|08/02/2027|Holmcroft
Way 87|London|05568|N/A|09/16/2010|12/31/9999|
0|276|Tomas|Wauber|11/17/1965|male|345678|12/05/1983|12/05/2013|Bacherstrasse
12, app.34|Munich|65078|N/A|09/18/2010|12/31/9999|
0|276|Yang|Lee|02/10/1979|male|456789|02/21/1999|02/21/2029|Shangtang Rd.
21|Hangzhou|12034|N/A|09/20/2010|12/31/9999|
0|040|Ingeborg|Rainer|03/25/1969|female|567890|04/11/1987|04/11/2011|Keiser
Ebersdorfer Str. 101|Vienna|32108|N/A|09/10/2010|12/31/9999|
0|804|Vladislav|Shevnya|10/13/1969|male|678901|06/01/1997|06/01/2021|Shevchenko
str. 71|Kyiv|21000|N/A|09/10/2010|12/31/9999|
0|840|Joseph|Williams|12/15/1972|male|789012|08/23/1992|08/23/2032|Washington
str. 23, app 15|Los Angeles|70504|N/A|09/12/2010|12/31/9999|
0|826|Dorothy|McMillan|05/16/1984|female|890123|09/30/1999|09/30/2039|Wellington
str. 92|Edinburgh|65008|N/A|09/10/2010|12/31/9999|
0|804|Maria|Rudenko|07/12/1981|female|9012345|06/21/2002|06/21/2042|Gogolya str.
25, app 85|Kharkiv|61000|N/A|09/15/2010|12/31/9999|
0|840|Stephany|Koch|09/03/1975|female|56789321|08/19/1993|08/19/2023|Jacksons
str. 40|Chicago|87543|N/A|09/10/2010|12/31/9999|
```

contracts.unl:

```
0|1|1|840|123|26302010001|5500.0|15.0|09/10/2010|12/31/9999|
0|1|1|840|4567|26304010002|20000.0|10.0|09/10/2010|12/31/9999|
0|2|2|840|732|26354010003|150000.0|12.0|09/10/2010|12/31/9999|
0|3|3|840|120|26303010003|150000.0|18.0|09/10/2010|12/31/9999|
0|1|4|826|386|26301070004|3000.0|10.0|09/10/2010|12/31/9999|
```

country_list.unl:

```
036|AUS|Australia|
```



040	AUT	Austria
031	AZE	Azerbaijan
008	ALB	Albania
012	DZA	Algeria
016	ASM	American Samoa
660	AIA	Anguilla
024	AGO	Angola
020	AND	Andorra
010	ATA	Antarctica
028	ATG	Antigua And Barbuda
032	ARG	Argentina
533	ABW	Aruba
004	AFG	Afghanistan
044	BHS	Bahamas
050	BGD	Bangladesh
052	BRB	Barbados
048	BHR	Bahrain
056	BEL	Belgium
084	BLZ	Belize
204	BEN	Benin
060	BMU	Bermuda
112	BLR	Belarus
100	BGR	Bulgaria
068	BOL	Bolivia
070	BIH	Bosnia And Herzegovina
072	BWA	Botswana
076	BRA	Brazil
086	IOT	British Indian Ocean Territory
096	BRN	Brunei Darussalam
854	BFA	Burkina Faso
108	BDI	Burundi
064	BTN	Bhutan
548	VUT	Vanuatu
862	VEN	Venezuela
704	VNM	Viet Nam
092	VGB	Virgin Islands (British)
850	VIR	Virgin Islands (U.S.)
051	ARM	Armenia
266	GAB	Gabon
328	GUY	Guyana
332	HTI	Haiti
270	GMB	Gambia
288	GHA	Ghana
312	GLP	Guadeloupe
320	GTM	Guatemala
324	GIN	Guinea
624	GNB	Guinea-Bissau
292	GIB	Gibraltar
340	HND	Honduras
344	HKG	Hong Kong
308	GRD	Grenada
304	GRL	Greenland
300	GRC	Greece
268	GEO	Georgia
316	GUM	Guam



208	DNK	Denmark
336	VAT	Holy See (Vatican City State)
262	DJI	Djibouti
212	DMA	Dominica
214	DOM	Dominican Republic
218	ECU	Ecuador
226	GNQ	Equatorial Guinea
232	ERI	Eritrea
233	EST	Estonia
231	ETH	Ethiopia
818	EGY	Egypt
887	YEM	Yemen
180	ZAR	Zaire
894	ZMB	Zambia
732	ESH	Western Sahara
716	ZWE	Zimbabwe
376	ISR	Israel
356	IND	India
360	IDN	Indonesia
368	IRQ	Iraq
364	IRN	Iran (Islamic Republic Of)
372	IRL	Ireland
352	ISL	Iceland
724	ESP	Spain
380	ITA	Italy
400	JOR	Jordan
132	CPV	Cape Verde
398	KAZ	Kazakhstan
116	KHM	Cambodia
120	CMR	Cameroon
124	CAN	Canada
634	QAT	Qatar
404	KEN	Kenya
417	KGZ	Kyrgyzstan
156	CHN	China
196	CYP	Cyprus
296	KIR	Kiribati
166	CCK	Cocos (Keeling) Islands
170	COL	Colombia
174	COM	Comoros
178	COG	Congo
408	PRK	Korea, Democratic People's Republic Of
410	KOR	Korea, Republic Of
188	CRI	Costa Rica
384	CIV	Cote D'Ivoire
192	CUB	Cuba
414	KWT	Kuwait
418	LAO	Lao People's Democratic Republic
428	LVA	Latvia
426	LSO	Lesotho
440	LTU	Lithuania
430	LBR	Liberia
422	LBN	Lebanon
434	LIB	Libyan Arab Jamahiriya
438	LIE	Liechtenstein



442	LUX	Luxembourg
480	MUS	Mauritius
478	MRT	Mauritania
450	MDG	Madagascar
175	MYT	Mayotte
446	MAC	Macau
807	MKD	Macedonia, The Former Yugoslav Republic Of
454	MWI	Malawi
458	MYS	Malaysia
466	MLI	Mali
581	UMI	United States Minor Outlying Islands
462	MDV	Maldives
470	MLT	Malta
504	MAR	Morocco
474	MTQ	Martinique
584	MHL	Marshall Islands
484	MEX	Mexico
583	FSM	Micronesia, Federated States Of
508	MOZ	Mozambique
498	MDA	Moldova, Republic Of
492	MCO	Monaco
496	MNG	Mongolia
500	MSR	Montserrat
104	MMR	Myanmar
516	NAM	Namibia
520	NRU	Nauru
524	NPL	Nepal
562	NER	Niger
566	NGA	Nigeria
528	NLD	Netherlands
530	ANT	Netherlands Antilles
558	NIC	Nicaragua
276	DEU	Germany
570	NIU	Niue
554	NZL	New Zealand
540	NCL	New Caledonia
578	NOR	Norway
834	TZA	Tanzania, United Republic Of
784	ARE	United Arab Emirates
512	OMN	Oman
074	BVT	Bouvet Island
574	NFK	Norfolk Island
162	CXR	Christmas Island
654	SHN	St. Helena
136	CYM	Cayman Islands
184	COK	Cook Islands
744	SJM	Svalbard And Jan Mayen Islands
876	WLF	Wallis And Futuna Islands
334	HMD	Heard And Mc Donald Islands
586	PAK	Pakistan
585	PLW	Palau
591	PAN	Panama
598	PNG	Papua New Guinea
600	PRY	Paraguay
604	PER	Peru



710	ZAF	South Africa
239	SGS	South Georgia And The South Sandwich Islands
580	MNP	Northern Mariana Islands
612	PCN	Pitcairn
616	POL	Poland
620	PRT	Portugal
630	PRI	Puerto Rico
638	REU	Reunion
643	RUS	Russian Federation
646	RWA	Rwanda
642	ROM	Romania
222	SLV	El Salvador
882	WSM	Samoa
674	SMR	San Marino
678	STP	Sao Tome And Principe
682	SAU	Saudi Arabia
748	SWZ	Swaziland
690	SYC	Seychelles
686	SEN	Senegal
659	KNA	Saint Kitts And Nevis
670	VCT	Saint Vincent And The Grenadines
662	LCA	Saint Lucia
666	SPM	St. Pierre And Miquelon
702	SGP	Singapore
760	SYR	Syrian Arab Republic
703	SVK	Slovakia (Slovak Republic)
705	SVN	Slovenia
090	SLB	Solomon Islands
706	SOM	Somalia
826	GBR	United Kingdom
840	USA	United States
736	SDN	Sudan
740	SUR	Suriname
626	TMP	East Timor
694	SLE	Sierra Leone
762	TJK	Tajikistan
764	THA	Thailand
158	TWN	Taiwan, Province Of China
796	TCA	Turks And Caicos Islands
768	TGO	Togo
772	TKL	Tokelau
776	TON	Tonga
780	TTO	Trinidad And Tobago
798	TUV	Tuvalu
788	TUN	Tunisia
792	TUR	Turkey
795	TKM	Turkmenistan
800	UGA	Uganda
348	HUN	Hungary
860	UZB	Uzbekistan
804	UKR	Ukraine
858	URY	Uruguay
234	FRO	Faroe Islands
242	FJI	Fiji
608	PHL	Philippines



```
246|FIN|Finland|
238|FLK|Falkland Islands (Malvinas)|
250|FRA|France|
249|FXX|France, Metropolitan|
254|GUF|French Guiana|
258|PYF|French Polynesia|
260|ATF|French Southern Territories|
191|HRV|Croatia (Local Name: Hrvatska)|
140|CAF|Central African Republic|
148|TCD|Chad|
203|CZE|Czech Republic|
152|CHL|Chile|
756|CHE|Switzerland|
752|SWE|Sweden|
144|LKA|Sri Lanka|
891|YUG|Yugoslavia|
388|JAM|Jamaica|
392|JPN|Japan|
```

currency.unl:

```
004|004|AFA|
008|008|ALL|
012|012|DZD|
020|020|ADP|
973|024|AOA|
031|031|AZM|
032|032|ARS|
036|036|AUD|
044|044|BSD|
048|048|BHD|
050|050|BDT|
051|051|AMD|
052|052|BBD|
060|060|BMD|
064|064|BTN|
068|068|BOB|
072|072|BWP|
084|084|BZD|
090|090|SBD|
096|096|BND|
100|100|BGL|
104|104|MMK|
108|108|BIF|
112|112|BYB|
116|116|KHR|
124|124|CAD|
132|132|CVE|
136|136|KYD|
144|144|LKR|
152|152|CLP|
156|156|CNY|
170|170|COP|
174|174|KMF|
```



188	188	CRC
191	191	HRK
192	192	CUP
196	196	CYP
203	203	CZK
208	208	DKK
214	214	DOP
222	222	SVC
230	231	ETB
233	233	EEK
238	238	FKP
242	242	FJD
262	262	DJF
270	270	GMD
288	288	GHC
292	292	GIP
320	320	GTQ
324	324	GNF
328	328	GYD
332	332	HTG
340	340	HNL
344	344	HKD
348	348	HUF
352	352	ISK
356	356	INR
360	360	IDR
364	364	IRR
368	368	IQD
376	376	ILS
981	268	GEL
388	388	JMD
392	392	JPY
398	398	KZT
400	400	JOD
404	404	KES
408	408	KPW
410	410	KRW
414	414	KWD
417	417	KGS
418	418	LAK
422	422	LBP
426	426	LSL
428	428	LVL
430	430	LRD
434	434	LYD
440	440	LTL
446	446	MOP
450	450	MGF
454	454	MWK
458	458	MYR
462	462	MVR
470	470	MTL
478	478	MRO
480	480	MUR
484	484	MXN



496	496	MNT
498	498	MDL
504	504	MAD
508	508	MZM
512	512	OMR
516	516	NAD
524	524	NPR
532	530	ANG
533	533	AWG
548	548	VUV
554	554	NZD
558	558	NIO
566	566	NGN
578	578	NOK
586	586	PKR
590	591	PAB
598	598	PGK
600	600	PYG
604	604	PEN
608	608	PHP
624	624	GWP
626	626	TPE
634	634	QAR
642	642	ROL
646	646	RWF
654	654	SHP
678	678	STD
682	682	SAR
690	690	SCR
694	694	SLL
702	702	SGD
703	703	SKK
704	704	VND
705	705	SIT
706	706	SOS
710	710	ZAR
716	716	ZWD
736	736	SDD
740	740	SRG
748	748	SZL
752	752	SEK
756	756	CHF
760	760	SYP
764	764	THB
776	776	TOP
780	780	TTD
784	784	AED
788	788	TND
792	792	TRL
795	795	TMM
800	800	UGX
807	807	MKD
818	818	EGP
826	826	GBP
834	834	TZS



838	795	TMR
839	417	KGR
840	840	USD
858	858	UYU
860	860	UZS
862	862	VEB
882	882	WST
886	887	YER
891	891	YUM
894	894	ZMK
901	158	TWD
974	112	BYR
978	276	EUR
980	804	UAH
985	616	PLN
986	076	BRL
990	152	CLF
999	643	RUS
972	762	TJS
976	178	CDF
232	232	ERN
977	070	BAM
643	643	RUB

type_contract.unl:

```
1|Short-term deposit|
2|Long-term deposit|
3|Call deposit|
```



Modifying Database Tables

To work with database tables you need not only to know how to create them and load values into them. You may also need to modify the structure of an already existing table - i.e. add or remove a column or a constraint. You may also want to delete a row from a table or add a new row of values supplied by the user and not predefined by the program, which cannot be done using the LOAD statement. All these activities can be managed using a set of embedded SQL statements discussed in this chapter.

Modifying the Table Structure

The ALTER TABLE statement is used to modify an already existing table. It is used to add a new column or delete an existing one; it can also modify the constraints placed on the columns. Thus you do not need to delete a column and create a new one, if the structure of your database requires changes. The syntax of this statement depends on the purpose it is used for.

Adding a Column

The syntax of the ALTER TABLE statement used to add a new column to a table is as follows:

```
ALTER TABLE table_name ADD (column clause)
```

Here the table name must belong to a table existing in the current database which you want to modify. There can be one column clause, if you want to add one column, or several column clauses, if you want to add several columns, which should be separated by commas.

The syntax of the column clause is very similar to the [column definition](#) of the CREATE TABLE statement, as the purpose of both is to create a column, but it has its peculiarities:

```
new_column data_type [DEFAULT clause] [constraint] [BEFORE column_name]
```

Here the “new column” stands for the name of the column you want to insert. You must specify its data type, it is the only obligatory clause just like for the column definition of the CREATE TABLE statement. You can also specify the DEFAULT value and the column-level constraint. All these clauses follow the same rules as the column definition of the CREATE TABLE statement.

The only difference is the BEFORE keyword. You can add the BEFORE keyword at the end of the column clause to specify the column before which the new column will be inserted. The “column name” following the BEFORE keyword must be the name of a column existing in the table.

The example below inserts the *l_name* column after the *f_name* column and before the *age* column:

```
CREATE TABLE pers_data(
  f_name CHAR(20),
  age INT
)
...
ALTER TABLE pers_data ADD(l_name CHAR(30) NOT NULL BEFORE age)
```



The following example adds two more columns after the age column as the BEFORE keywords are absent. Thus the table will contain five columns in such order: *fname, lname, age, gender, address*.

```
ALTER TABLE pers_data ADD(gender CHAR, address CHAR(100))
```

Deleting a Column

The syntax of the ALTER TABLE statement used to delete a column from a table is as follows:

```
ALTER TABLE table_name DROP (column_name)
```

Here the table name must belong to a table existing in the current database which you want to modify. You can specify one column name in the parentheses, if you want to drop only one column. If you want to drop several columns, specify several column names separated by commas.

The column you want to drop must exist in the table. All the constraints placed on this column are dropped. If the deleted column is UNIQUE or PRIMARY KEY, the foreign keys referencing it from the other tables will be also dropped.

Here is an example of a columns being dropped:

```
ALTER TABLE pers_data DROP(gender, address)
```

Thus we dropped the columns previously added by the ADD clause in the section above.

Modifying an Existing Column

The syntax of the ALTER TABLE statement used to modify an existing column in a table is as follows:

```
ALTER TABLE table_name MODIFY (column clause)
```

Here the table name must belong to a table existing in the current database which you want to modify. There can be one column clause, if you want to modify one column, or several column clauses, if you want to modify several columns, which should be separated by commas. The syntax of the column clause is the same as the syntax of the [column definition](#) in the CREATE TABLE statement:

```
new_column data_type [DEFAULT clause] [constraint]
```

You can use this statement to modify any of the column attributes except its name. When a column is used in the MODIFY clause, all the attributes including the data type, default value and constraints are dropped. If you want the modified column to retain any of its attributes, you need to re-specify them in the column clause.

The example below modifies one of the table columns adding the default value and placing a constraint:

```
CREATE TABLE pers_data(
  f_name CHAR(20),
  l_name CHAR(30),
  age INT
)
...
ALTER TABLE pers_data MODIFY(age INT DEFAULT "1" NOT NULL)
```



Modifying Constraints

Using the ALTER TABLE statement you can enable or disable an existing constraint or create a new constraint.

The syntax of the ALTER TABLE statement used to add a constraint on a column in a table is as follows:

```
ALTER TABLE table_name ADD CONSTRAINT (table-level_constraint)
```

You can manipulate only table level constraints using this statement. The table level constraints which can be added to a column are the same which are described in the [“Table-level constraints”](#) of the CREATE TABLE statement. E.g.:

```
ALTER TABLE my_tab_2
    ADD CONSTRAINT (FOREIGN KEY (ref_id)
                    REFERENCES my_tab_1(user_id)
                    CONSTRAINT fk_user_id)
```

To drop an existing constraint, use the following syntax:

```
ALTER TABLE table_name
    DROP CONSTRAINT (constraint_name [,constraint_name])
```

The constraint name is the name of the constraint given to it when the table was created. For more details see the [“Naming constraints”](#) section of the CREATE TABLE statement description. Below is an example of dropping a constraint.

```
ALTER TABLE my_tab_2
    DROP CONSTRAINT (fk_user_id, u_us_des)
```

Inserting Rows Into a Table

The insert statement is used to insert a row of values into a table. As a rule it is used to insert a user supplied values into a database table. Its general syntax is as follows:

```
INSERT INTO table_name [(column_list)] VALUES (values_list)
```

Here the table name is the name of the table into which you want to insert some values. The column list is optional and is required only if you insert the values not into all the columns of the table but only into some of them. The column names in the column list should be separated by commas. The values list contains the values to be inserted; it is discussed in details below.

The VALUES Clause

The values clause consists of the VALUES keyword and the values enclosed in parentheses. The values should be separated by commas from one another. The total number of values in the parentheses should correspond either to the total number of columns in the specified table, if the column list is empty, or to the



number of columns specified in the column list. The values data types should correspond to the data types of the columns they should be inserted it.

A value in the list may be represented by the following:

- A variable containing the value you want to insert. The variable and the corresponding column must be of compatible data types.
- A literal value (a literal number, DATETIME, INTERVAL)
- A quoted character string
- The TODAY operator to be inserted into a DATE column
- The CURRENT operator to be inserted into a DATETIME column

Inserting Values into a SERIAL Column

If you want to insert values into a column of [SERIAL](#) data type, enter a zero value for this column in the VALUES clause. When a zero is inserted into a SERIAL column, the database server assigns the next highest value to that column instead of zero. If you want to specify the SERIAL value explicitly, be sure that the number you insert does not duplicate one of the existing serial numbers in that column, otherwise an error will occur.

Here is an example of two INSERT statements:

```
CREATE TABLE my_tab(
    col_1 SERIAL,
    col_2 CHAR(60),
    col_3 INT,
    col_4 MONEY,
    col_5 DATE
)
...
LET char_v = "This is a value to be inserted"
INPUT date_v, money_v FROM f001, f002
...
INSERT INTO my_tab VALUES (0, char_v, int_v, money_v, "07/23/2010")
...
INSERT INTO my_tab (col_3, col_4, col_5)
    VALUES (int_v, 145.67, TODAY)
```

The VALUES clause of the first one contains a value for every column in the table and thus the column list is omitted. The VALUE clause of the second one contains only three values so the column clause is included. The values are partially retrieved from the user input. The code above will result in two rows of values added to the table.

Inserting Values Selected from Another Column

You can insert values selected from another column without having to execute the SELECT statement before the INSERT statement. For this purpose the VALUES clause can be substituted for the SELECT clause similar to the [SELECT clause](#) of the UNLOAD statement. The INSERT statement used to insert values from one table into another has the following syntax:

```
INSERT INTO table_name [(column_list)] SELECT clause
```



The SELECT clause in its turn must not contain the INTO clause and should look as follows:

```
SELECT select_list FROM table_list [WHERE clause]
```

The select list contains the names of the columns contained by the table specified in the FROM clause. It can also be represented by an asterisk. However, the number, order, and data types of the columns in the select list and in the column list of the INSERT statement must match. Thus, if you use an asterisk in the select list, you must pay attention to the number of columns in the table from which the values are selected.

The table list can contain more than one table name, they should be separated by commas.

The SELECT clause copies all the values which meet the WHERE condition from the specified table and inserts them into the table specified after the INSERT INTO keywords. If there is no WHERE clause and you specify an asterisk instead of the select list all the rows will be copied.

Here is an example illustrating the above mentioned issues:

```
CREATE TABLE my_tab1(
    col_11 CHAR(60),
    col_12 INT,
    col_13 MONEY
)

CREATE TABLE my_tab2(
    col_21 SERIAL,
    col_22 CHAR(60),
    col_23 INT,
    col_24 MONEY,
    col_25 DATE
)
.....
INSERT INTO my_tab SELECT col_22, col_23, col_24
    FROM my_tab2 WHERE col_22 MATCHES "*st*"
...
INSERT INTO my_tab2 (col_22, col_23, col_24)
    SELECT * FROM my_tab WHERE col_24 > 100.00
```

The first INSERT statement does not include the column list, the table has only three columns and the SELECT clause selects three values. For the second INSERT statement the column list is necessary, because column "my_tab" has fewer columns than "my_tab2" and the SELECT clause would not have been able to supply values for all the columns of "my_tab2".

Deleting Rows From a Table

The DELETE statement is used to delete one or more rows from a table. It has the following syntax:

```
DELETE FROM table_name [WHERE condition]
```

As the DELETE statement is used to delete rows and not columns, you cannot specify the row you want to delete explicitly, because the rows do not have names or identifiers. However, you can delete a row by specifying the condition which should meet one or more of its values. The WHERE clause is used to specify which row or rows you want to delete.



The WHERE condition is optional. If it is absent, all the rows are deleted from the table specified after the DELETE FROM keywords.

The WHERE Clause

The WHERE clause is used to specify the rows you want to delete. The rules for the WHERE condition are the same as for the [WHERE clause](#) of the SELECT statement. A row is deleted, if it meets the condition specified in the WHERE clause.

The example below is used to delete all rows which contain values less than 1 in the "amount" column:

```
DELETE FROM goods_list WHERE amount < 1
```

As well as in the WHERE clause of the SELECT statement you can combine several conditions with the help of the Boolean operators AND and OR in order to make the deletion more precise, e.g.:

```
DELETE FROM pers_info WHERE f_date < TODAY AND id_stat MATCHES "closed"
```

Modifying the Existing Rows

The UPDATE statement is used to change the values in the rows which already exist in the table. The execution of this statement will not result in a new row added to the table. It can be used to change the value of one column in a row or of several columns. This statement has the following syntax:

```
UPDATE table_name SET clause [WHERE clause]
```

The table name is the name of the table in which you want to update a row or rows. The SET clause contains the column names and the new values for the specified columns which would be inserted instead of the old values. The WHERE clause is optional, it specifies which row to update.

If the WHERE clause is absent, the values of all the rows in the specified columns will be changed.

The SET Clause

The SET clause specifies the values for which columns in the row must be changed and provides the new values. Its syntax is as follows:

```
SET column_name = value [, column_name = value...]
```

The "column name" is the column the value in which you want to change. A SET clause can specify the values for several columns, the column names together with the new values should be separated by a comma from each other.

The "value" is the new value to be inserted into the specified column for the row which meets the WHERE condition. It can be represented by any 4GL expression, a variable or a literal, provided that the value returned by this expression matches the data type of the specified column. You can also use the NULL keyword to assign a null value to a column in the row.



The example below will change all the value of the “total” column in all the rows to the value returned by the expression:

```
CREATE TABLE orders(
    price MONEY,
    quantity INT,
    total DEC
)
...
UPDATE orders SET total = price*quantity
```

The value assigned to “total” column in each row will be the result of the values in “price” and “quantity” columns of that row multiplied. Each row will contain a different value for this column which will depend on the “price” and “quantity” values for this very row.

The example below uses literals as the values to be assigned:

```
UPDATE orders SET      price = 100.90,
                      quantity = 2,
                      total = 2*100.90
```

Using Values from other Tables for Update

It is possible to use the variables from other tables instead of the values represented by 4GL expressions. Like with the INSERT statement, the UPDATE statement can use the SELECT clause. The values retrieved by the SELECT clause are assigned to the specified columns. In this case the SELECT clause is included into the SET clause instead of the “value” section. The syntax is as follows:

```
SET column_name = (SELECT select_list FROM table_name [WHERE condition])
```

The SELECT clause follows the rules specified for the stand-alone SELECT statement. The table name here specifies the table from which you want to retrieve the values. The WHERE condition specifies the condition under which the values will be retrieved, do not mix it up with the WHERE clause of the UPDATE statement itself, which specifies the condition which should be met by the rows which are to be modified.

Here is an example of the update STATEMENT with the SELECT clause:

```
UPDATE orders
    SET item_name = (SELECT item_name FROM item_list
                     WHERE orders.item_id = item_name.item_id)
```

Other Ways to Specify the Columns to Update

You can also specify the column names and the values to be assigned to them in another way, if you want to change the values of more than one column:

```
SET (column_name [, column_name]) = (value [, value])
```

You can also specify the asterisk (*) instead of the column list which will indicate that all the columns of this row must be updated:

```
SET * = (value [, value])
```



In this case you must pay attention to the order of the values which should correspond to the order of the columns in the table to be modified.

```
UPDATE pers
    SET * = ("N/A", "N/A")
  WHERE lname = "Smith"
```

The WHERE Clause

The WHERE clause is used to specify the rows you want to update. The rules for the WHERE condition are the same as for the [WHERE clause](#) of the SELECT statement. A row is updated with the new values, if it meets the condition specified in the WHERE clause. You should not mix this WHERE clause with that of the SELECT clause.

Here are several examples of the UPDATE statement with the WHERE clause:

```
UPDATE orders
    SET item_name = (SELECT item_name FROM item_list
                      WHERE orders.item_id = item_name.item_id)
        WHERE item_id > 500

UPDATE orders
    SET item_descr = (SELECT descr FROM item_list
                      WHERE orders.item_id = item_name.item_id),
        item_cond = "available"
        WHERE item_id > 500 OR item_id < 100
```

As you can see, these are two WHERE clauses in each example one belonging to the UPDATE statement proper and the second as a part of the SELECT clause.

Example

This example illustrates how a database table, once created and loaded, can be modified. The modifications can occur both in the structure of a table and in the data stored in it.

```
#####
# This examples modifies the structure of the tables and the data in them
#####
DATABASE db_examples

DEFINE r_clients RECORD -- the structure of the record is the same as the
                      -- structure of clients table
    id_client      INTEGER,
    id_country     CHAR(3),
    first_name     CHAR(20),
    last_name      CHAR(20),
    date_birth     DATE,
    gender         CHAR(6),
    id_passport    CHAR(10),
    date_issue     DATE,
    date_expire    DATE,
    address        CHAR(80),
```



```

        city      CHAR(20),
        zip_code CHAR(6),
        add_data CHAR(512),
        start_date DATE,
        end_date   DATE
    END RECORD,
r_tmp_clients RECORD
    id_client  INTEGER,
    id_country CHAR(3),
    first_name CHAR(30),
    last_name  CHAR(30),
    id_passport CHAR(10),
    address    CHAR(80),
    start_date DATE,
    end_date   DATE
END RECORD

MAIN

MENU "Altering"
    COMMAND "Altering Tables"
        "Adding columns to tables, changing column properties, etc."
        CALL modify_tables()
    COMMAND "Reverting Changes"
        "Reverts the changes made by the ALTER statements"
        CALL restore()
    COMMAND "Inserting Rows"
        "Inserts new rows of data into the tables"
        CALL insert_rows()
    COMMAND "Modifying Rows"
        "Changes the values already recorded in the tables"
        CALL modify_rows()
    COMMAND "Deleting Rows"
        "Deletes some of the rows from tables"
        CALL delete_rows()

    COMMAND "EXIT"
        # Before we exit from the program we need to
        # restore the database state to be able to use this
        # database for the future examples
        CALL restore()
    EXIT PROGRAM
END MENU

END MAIN

#####
# This function adds columns, deletes constraints and
# changes the characteristics of the columns
#####
FUNCTION modify_tables()
    CLEAR SCREEN

    # First we remove constraints created in previous sample applications
    # The names of the constraints were given when they were created in

```



```

# the CREATE TABLE statement

ALTER TABLE contracts DROP CONSTRAINT r_contract_id_type
ALTER TABLE clients   DROP CONSTRAINT c_clients_gender

#Then we add new columns into the clients table

# The last_column CHAR(80) column will be added after
#the last column present which is end_date
ALTER TABLE clients ADD(last_column CHAR(80))

# The new_column INTEGER column will be added between
# columns add_data and start_date

ALTER TABLE clients ADD(new_column INTEGER BEFORE start_date)

# Then we change the size and check again:
# The size of the add_data column is changed from 512 to 256
ALTER TABLE clients MODIFY (add_data CHAR(256))

# The attribute of the address column is changed from
# "CHAR(80) NOT NULL" to CHAR(80)
# Thus we removed the NOT NULL constraint.
ALTER TABLE clients MODIFY (address CHAR(80))

CALL fgl_winmessage ("Tables Modifications",
"The tables and column constraints have been modified." ||
"\nYou can see the modified structure using the native DB tool" ||
"\nbefore you revert the changes or exit the application.", 
"info")

END FUNCTION

#####
# This function uses the ALTER TABLE statement to restore the state
# of the database as it was at the beginning of the application
#####
FUNCTION restore()
    CLEAR SCREEN
    WHENEVER ERROR CONTINUE -- in case there is nothing to revert
                                -- we handle the possible error

    #Here we will restore the database structure modified.

    # We remove the additional columns last_column and new_column inserted
    # into the table earlier.
    ALTER TABLE clients DROP (last_column,new_column)
    # We restore the attributes of the columns
    ALTER TABLE clients MODIFY (add_data CHAR(512))
    ALTER TABLE clients MODIFY (address char(80) NOT NULL)

    # We recreated the constraints removed earlier
    ALTER TABLE contracts ADD CONSTRAINT

```



```
(FOREIGN KEY (id_type) REFERENCES type_contract
CONSTRAINT r_contract_id_type)
# pay attention to the syntax of the CHECK constraint
# created by the ALTER TABLE statement
ALTER TABLE clients ADD CONSTRAINT
  (CHECK (gender IN ("male", "female")))
CONSTRAINT c_clients_gender)

CALL fgl_winmessage("Database Restored",
  "The initial state of the database was restored",
  "info")
WHENEVER ERROR STOP
END FUNCTION

#####
# This function inserts new data into tables and displays the
# inserted values
#####
FUNCTION insert_rows()
  WHENEVER ERROR CONTINUE
  CLEAR SCREEN
  INSERT INTO clients -- we insert values into all table columns,
    -- because the column list is absent
  VALUES (0, 804, "Julian", "Menson", "01/01/1980", "male", "12345",
    "10/01/1998", "10/01/2018", "Broadway. 100, app.56", "San Francisco",
    NULL, "N/A", "09/14/2010", "12/31/9999")

  # Then we insert another row containing only values
  # for the columns in the column list
  INSERT INTO clients (id_client, id_country, first_name,
    last_name, id_passport, address, start_date, end_date)
  VALUES (0, 804, "Jim", "Holiday", "87654321", "Burbon 12, app.4",
    "09/14/2010", "12/31/9999")

  IF status = -268 THEN -- this value occurs when the constraint condition
    -- is violated; this may happen if you use this option
    -- more than once before deleting the inserted rows
    CALL fgl_winmessage("Insert Failed",
      "The rows with the given values already exist in the table" ||
      "\nyou need to delete them first before inserting again",
      "exclamation")
  WHENEVER ERROR STOP
  RETURN -- we leave this function
END IF

# Then we retrieve the newly inserted rows and display them
SELECT c.* INTO r_clients.* FROM clients c WHERE c.id_passport = "12345"
DISPLAY "The first inserted record with passport_id = 12345: "
DISPLAY r_clients.*
SELECT c.* INTO r_clients.* FROM clients c WHERE c.id_passport = "87654321"
DISPLAY "The second inserted record with passport_id = 87654321",
  "(some columns are empty): "
DISPLAY r_clients.*

WHENEVER ERROR STOP
```



```
END FUNCTION

#####
# This function modifies the data in the existing rows
#####
FUNCTION modify_rows()
    CLEAR SCREEN
    # We create the temporary table tmp_clients which will be used for
    # illustrating modifications
    # To avoid interfering with the data of the permanent table.
    CREATE TEMP TABLE tmp_clients -- the table has fewer columns than its
                                    -- prototype
    (
        id_client      INTEGER,
        id_country     CHAR(3),
        first_name     CHAR(30),
        last_name      CHAR(30),
        id_passport    CHAR(10),
        address        CHAR(80),
        start_date     DATE,
        end_date       DATE
    )
    # We populate the temporary table with some values
    # retrieved from the permanent table
    INSERT INTO tmp_clients
    SELECT c.id_client,
           c.id_country,
           c.first_name,
           c.last_name,
           c.id_passport,
           c.address,
           c.start_date,
           c.end_date
    FROM clients c

    # We display the row with id_client = 3 before any modifications are made
    SELECT c.* INTO r_tmp_clients.*
        FROM tmp_clients c   -- pay attention, we use the temporary table here
        WHERE c.id_client = 1

    DISPLAY "One of the rows before updating id_country (with client_id=1): "
    DISPLAY r_tmp_clients.*

    # We assign value "804" in the id_country for all the rows
    # as there is no WHERE clause
    UPDATE tmp_clients SET tmp_clients.id_country = "804"

    SELECT c.* INTO r_tmp_clients.*
        FROM tmp_clients c WHERE c.id_client = 3

    DISPLAY "The same row after updating (all rows now have 804 for country_id):"
    DISPLAY r_tmp_clients.*
```



```

# The values of columns first_name, last_name, address, id_passport
# will be edited only in one row because the WHERE clause is present
UPDATE tmp_clients SET first_name = "Andre",last_name = "Jarr",
                      id_passport = "1002450",address = "New address of Andre Jarr"
WHERE id_client = 3

SELECT c.* INTO r_tmp_clients.*
  FROM tmp_clients c WHERE c.id_client = 3
DISPLAY "The row with client_id = 3 again after updating some columns: "
DISPLAY r_tmp_clients.*


SELECT c.* INTO r_tmp_clients.*
  FROM tmp_clients c WHERE c.id_client = 2
DISPLAY "The row with client_id = 2 before further updating ",
        "(but with country_id = 840 already): "
DISPLAY r_tmp_clients.*

# The columns id_country and start_date will be changed
# in those rows where id_client is 2,4, and 5
UPDATE tmp_clients SET id_country = "040",start_date = TODAY
WHERE id_client IN (2,4,5)
SELECT c.* INTO r_tmp_clients.*
  FROM tmp_clients c WHERE c.id_client = 2
DISPLAY "The row with client_id=2 after updating id_country and start_date: "
DISPLAY r_tmp_clients.*


# The columns id_country and start_date will be modified in the rows with
# id_client having value from 2 to 5 inclusive
UPDATE tmp_clients SET (id_country,start_date) = ("818",TODAY)
WHERE id_client BETWEEN 2 AND 5

SELECT c.* INTO r_tmp_clients.*
  FROM tmp_clients c WHERE c.id_client = 2
DISPLAY "Another subsequent modification of row with client_id = 2:"
DISPLAY r_tmp_clients.*


END FUNCTION

#####
# This function deletes some rows from database tables
#####
FUNCTION delete_rows()
    CLEAR SCREEN
    WHENEVER ERROR CONTINUE
    #First we delete rows from the temporary table

    # This statement deletes only one row which meets the WHERE condition
    # We use this statement to verify whether the temporary table exists
    DELETE FROM tmp_clients WHERE id_client = 3

    IF status = -206 THEN -- this value means that the table was not found
        CALL fgl_winmessage( "Temporary Table",
                            "The temporary table does not exits" || )
    END IF

```



```
\nIt may happen because you restarted the application" ||
"\nand didn't recreate it. Run ""Modify Rows"" option" ||
"\nto recreate the temporary table.", "info")
ELSE
# If the table exists we check whether it contains any rows
SELECT c.* INTO r_tmp_clients.*
FROM tmp_clients c WHERE c.id_client = 1
IF status = 100 THEN
    CALL fgl_winmessage("Deleting Failed",
    "There are no row in the temporary table to be deleted", "info")
ELSE
    # The rows with id_client containing values from 2 to 5 will be
    # also deleted
    DELETE FROM tmp_clients WHERE tmp_clients.id_client BETWEEN 3 AND 5
    # The statement below deletes all the rows from a table leaving it
    # empty
    DELETE FROM tmp_clients
    SELECT c.* INTO r_tmp_clients.*
    FROM tmp_clients c WHERE c.id_client = 1
    IF status = 100 THEN
        CALL fgl_winmessage("Deleting",
        "Deleting from temp_clients table SUCCEEDED", "info")
    ELSE
        CALL fgl_winmessage("Deleting",
        "Deleting from temp_clients table FAILED:" ||
        "\nsuch rows still exist.", "exclamation")
    END IF
END IF

END IF

# We need to check whether additional rows exist in the table
SELECT c.* INTO r_tmp_clients.*
FROM clients c WHERE c.id_client > 10
IF status = 100 THEN
    CALL fgl_winmessage("Rows are Absent",
    "There are no rows which can be deleted" ||
    "\nfrom table clients. Run the ""Insert Rows"" option" ||
    "\n to insert the rows which then can be deleted.", "exclamation")
ELSE
    # We delete the rows from the permanent table which id_client is greater
    # than 10.
    # Thus we will delete all the rows added by the INSERT statements in this
    # module.
    DELETE FROM clients WHERE clients.id_client > 10
    # Now we check again whether they exist
    SELECT c.* INTO r_tmp_clients.*
    FROM clients c WHERE c.id_client > 10
    IF status = 100 THEN
        CALL fgl_winmessage("Deleting",
        "The rows were deleted from clients table",
        "info")
    ELSE
        CALL fgl_winmessage("Deleting",
        "Deleting from clients table FAILED:" ||
```



```
        "\nsuch rows still exist.", "exclamation" )  
    END IF  
END IF  
  
WHENEVER ERROR STOP  
END FUNCTION
```



Variables and Forms Adjusted for Being Used with Databases

When designing an application which requires a database connection, you can use some special statements to facilitate the variables declaration and values assignment. There are also some features which can be applied to form files to make the field declaration more convenient.

All the statements and features listed below are applied at compile time and require a default database to be declared. Any association between variables (or form fields) and database tables is possible only if these tables exist permanently in the database. Thus you cannot establish a link between a variable (or a form field) and a column of a temporary table. The temporary tables exist only during the runtime of your application whereas the features described in this chapter are applied during compilation.

The values in the syscolatt and syscolval system tables mentioned further in this chapter are managed with the help of the statements used to manage table contents.

Views

A view is a set of columns filled with values which is based upon the tables existing in the database. A view behaves like a table. It consists of the rows and columns which are selected from a column or columns. Thus a view can combine information from several tables, or it can be a restricted variant of one table, if it contains only some of its columns and/or rows.

A view can be created using the CREATE VIEW statement:

```
CREATE VIEW view_name [(column list)] AS SELECT clause
```

The view name can be used instead of the table names in most of the SQL statements except CREATE TABLE, ALTER TABLE, DROP TABLE and any others used to create or modify tables. For example, if it is used in a SELECT statement, you can select values from a view.

If some rows are added to or removed from the tables on the basis of which the view was created, the corresponding changes are made in the view automatically. If some rows are added to or deleted from the view, it affects the underlying tables likewise. However, if the underlying tables are modified using the ALTER TABLE statement and some columns are added it will not be reflected in the view.

The SELECT Clause

The SELECT clause of the CREATE VIEW statement has the form described for the SELECT clause of the UNLOAD statement and of the INSERT statement. This SELECT clause does not contain the INTO sub-clause and has the following syntax:

```
SELECT select_list FROM table_name [WHERE clause]
```

The select list can include not only column names but also arithmetic expressions which include column names, e.g.:

```
SELECT orders.price*1.05...
```



If such element is selected into a view, it is assigned to a so called virtual column. The section below contains the details on the view columns in general and virtual columns in particular.

The columns specified in the select list must be preceded by the table names, if the view is based on more than one table and the columns names are not unique among those tables.

The Column List

The column list is optional, it is included, if you want the view to have columns with names that differ from the columns returned by the SELECT clause. If the column list is omitted, the columns of the view inherit the column names of the underlying tables. The column names in the column list must correspond in number to the columns retrieved by the SELECT clause.

If the SELECT statement returns not a column but an expression, the column in the view which acquires this expression is called a virtual column. You must provide the name for such column in the column list.

Consequently, if you must provide the name for at least one column, you must provide the names for the rest of the columns as well.

The view created by the example below will have three columns which names will be the same as those of the selected columns. All the rows will be selected for the specified columns, as the WHERE clause is absent. However, if some columns are added to those tables after the view is created, they will not be added to the view:

```
CREATE VIEW my_view1 AS
    SELECT items.item_id, items.item_price, orders.items_free
    FROM items, orders
```

The example below creates a view based on two tables. As the select list contains an arithmetic expression, the column list is required. The WHERE sub-clause indicates, that not all the rows for the specified columns are added to the view:

```
CREATE VIEW my_view1 (ids, prices, left) AS
    SELECT      items.item_id,           items.item_price*1.15,
    orders.items_free
    FROM items, orders
    WHERE items.items_id = orders.items_id AND items.items_id
>100
```

Dropping the View

You must have the DBA privileges to be able to drop a view. When a view is dropped, it is removed from a database. However, it does not affect the tables on which it has been created. The syntax of the DROP VIEW statement is as follows:

```
DROP VIEW view_name
```

The view name must belong to a previously created view. If such view does not exist in the database, an error will occur.



Synonyms

A synonym provides an alternative name for a table or a view. The synonym name must be unique among other tables, views and synonyms of the same database. It is created by the CREATE SYNONYM statement, here is its syntax:

```
CREATE SYNONYM synonym_name FOR table_name
```

You can create synonyms not only for the tables in the same database, but also for the tables located in other databases, provided that they are on the same database server. You need to use the "owner" prefix, if the database is ANSI-compliant. Below, there is an example of a synonym created for a table in another database (not in the database which is the current database for this application). It is valid, provided that your qx_db and qx_test_db databases are located on the same database server. It creates the synonym for the table called "my_tab" owned by "me"

```
DATABASE qx_db
...
CREATE SYNONYM my_syn FOR qx_test_db:me.my_tab
```

Thus it allows you to reference tables from a non current database quickly and efficiently.

Dropping Synonyms

The synonym exists in the database once created as well as views and permanent tables. To delete a synonym, you should use the DROP SYNONYM statement, its syntax is as follows:

```
DROP SYNONYM synonym_name
```

If a table, for which a synonym was created, is dropped and they are both in the same database, the corresponding synonym will also be dropped.

Declaring Variables Linked to Table Columns

The DEFINE statement which we already discussed and used has a form allowing you to declare a variable LIKE a table column. Declaring a variable in such a way, you indicate that it has the same data type and precision as the column you specify.

The syntax of the DEFINE statement used to link variables to database columns is as follows:

```
DEFINE variable LIKE table.column
```

The table.column specification can be preceded by the owner name and/or database name.

You can use this format of the DEFINE statement only if you have the [default database](#) specified. Thus you need a DATABASE statement at the beginning of any program module containing a DEFINE ... LIKE statement. If the DEFINE ...LIKE statement is used to declare a global variable, the DATABASE statement must precede the GLOBALS keyword in the GLOBALS file.

A variable declared LIKE a table column will have any simple or large data type depending on the data type of the column. The example below declares a variable like "column1". Provided that this column is of CHAR(15) data type, the variable will also be of CHAR data type:

```
DEFINE test_var LIKE my_tab.column1
```



This method of declaration is useful when working with the database data - you do not need to worry about the data type compatibility.

RECORD Variables Linked to Database Tables

The variable used in the DEFINE...LIKE statement can be either a simple variable or a record member, e.g.:

```
DEFINE my_rec RECORD
    memb 1 INT,
    memb2 LIKE my_tab.col2
END RECORD
```

If you want to declare a record whose members correspond in number, names, and data types to all the columns in a database table, you can use the following syntax:

```
DEFINE record_name RECORD LIKE table.*
```

	Note: The END RECORD keywords in this case are omitted.
---	--

In such case the record will contain as many members as the table has columns. For example, table "orders" can be declared as follows:

```
CREATE TABLE orders(
    item_id CHAR(20),
    item_amount INT,
    price MONEY,
    total DEC
)
```

Then the two ways of declaring program record "my_pr_rec" shown below will produce the same effect:

```
DEFINE my_pr_rec RECORD LIKE orders.*
...
DEFINE my_pr_rec RECORD
    item_id CHAR(20),
    item_amount INT,
    price MONEY,
    total DEC
END RECORD
```

Validating a Value Against a Database Column

The VALIDATE statement checks whether the variable value is within the range of values allowed for a table column. This statement is useful when the values for tables are supplied via the input. Thus it helps to ensure that the value entered by a user fits the column in which it should be inserted before the insert is actually performed. It helps to avoid errors during inserting.



The syntax of the VALIDATE statement is as follows:

```
VALIDATE variable [,variable] LIKE table.column [, table.column]
```

The table.column identifier must belong to an existing column in a database. To be able to validate a value 4GL should refer to the [default database](#). Thus the DATABASE statement must occur before the first program block in the module where the VALIDATE statement is used, even if you add the database prefix before the table.column identifier. If you want to validate the values against all the columns of a table, you can use the table.* method.

The “variable” can be either a variable of a simple data type, a whole program record (record.*), its member (record.member), or a subset of record members identified with the help of the THRU/THROUGH keyword (record.member1 THRU record.member5). It can also be an element of a program array, but not the array as the whole (array[2,3]). However, you cannot validate the value of a TEXT or BYTE variable.

Usage

When some data are entered by a user into a form field, these data are automatically checked against the form field attributes. In such case you can check the validity of these data before inserting them into a database table. The form field attributes used to perform that check are discussed later in this chapter.

However, if your program inserts data into a database table from sources other than screen form (i.e. received with the help of the PROMPT statement), you can use the VALIDATE statement to perform the check. This statement has no effect unless INCLUDE values in the syscolval table are assigned for at least one of the database columns in the column list of the VALIDATE statement.

If the validation fails and the value of a variable does not correspond to the validation criteria, the VALIDATE statement sets the built-in **status** variable to -1321. If the verification is successful, the **status** variable is assigned 0. If you specify more than one variable in the VALIDATE statement (record.* and record members subsets are treated as multiple variables) and receive a negative value of the **status** variable, you need to verify each variable separately to identify the one which does not meet the validation criteria.

The variables must match the columns specified in the LIKE clause in order, number, and data types. Here is an example of a VALIDATE statement:

```
VALIDATE my_var1, my_var2, my_rec.*  
      LIKE tab1.col1, tab1.col4, tab2.*
```

Initializing Variables with Default Column Values

You can assign a default value of a table column to a variable with the help of the INITIALIZE statement. We have already discussed this statement used with the TO NULL keywords. The INITIALIZE statement with the LIKE clause has the following syntax:

```
INITIALIZE variable [,variable] LIKE table.column [, table.column]
```

The “variable” can be either a variable of a simple data type, a whole program record (record.*), its member (record.member) or a subset of record members identified with the help of the THRU/THROUGH keyword (record.member1 THRU record.member5). It can also be an element of a program array, but not the array as the whole (array[2,3]). However, you cannot validate the value of a TEXT or BYTE variable.



The table.column identifier can be preceded by the database name or the owner name. It has to be a column of an existing database table. If you want to assign the default values of all columns in a table to a list of variables, you can use the table.* indicator.

The INITIALIZE...LIKE statement requires the default database to be declared with the help of the DATABASE statement in the same manner as the DEFINE ...LIKE and VALIDATE statements do. If no default database is specified or if it is inaccessible, you will receive a compile-time error.

The Default Values

The INITIALIZE ...LIKE statement can assign only default values of table columns. 4GL looks up the default values for database columns in the DEFAULT column of the *syscolva*/table in the default database. Any changes to *syscolva*/after compilation have no effect on the 4GL program, unless you recompile the program. To enter default values in this table, use the *upscol* utility.

Here is an example of the INITIALIZE...LIKE statement in use:

```
INITIALIZE my_rec.* LIKE my_tab.*
```

Form Elements Linked to a Database

A form file can be adjusted for being used with a database. Even without all the adjustments mentioned below a form file can be used to display information received from a database or to get the information to be inserted into it. However, there are some features which facilitate the process.

A form file modified to work with a database more efficiently usually has the following features included:

- The DATABASE keyword at the beginning of the form file is followed by the database the form is supposed to be compiled against and not by the *formonly* keyword. This is obligatory if any of the following is included.
- The form fields used to input or display data connected with a database are linked to the corresponding table columns.
- The TABLES section is included into the form specification where all the tables, to which the form fields are linked, are listed.
- The form fields may acquire some special attributes, such as VALIDATE, which make them depend on a database column.

Declaring a Non-Formonly Screen form

If you plan to link any of the form fields to database columns, you cannot leave the form as *formonly*. The DATABASE keyword at the beginning of the form file linked to a database is used for the same purpose as the DATABASE statement in a 4GL source file - to specify the default database against which the file is to be compiled.

Thus the DATABASE keyword should be followed by the database name which contains the columns you want to link the form fields to, e.g.:

```
DATABASE my_db
SCREEN{
...
}
```



Even if there is a database declared by the DATABASE statement, you can still declare form fields as being formonly.

The TABLES Section

The TABLES section must be placed immediately after the SCREEN section and before the ATTRIBUTES section. This section must contain the names of the tables from the database specified by the DATABASE statement above which you will use. If there are more than one table in the list, you should not separate them by commas or any other symbols.

```
DATABASE my_db
SCREEN{
...
}
TABLES
col1 col2 col3
```

You can also use an alias instead of the table name, if that name is too long and inconvenient. Use the following syntax to declare an alias instead of just specifying a column name:

```
alias=[database.owner.]table_name
```

Here the alias is the keyword which will replace the table specified in this form file. The alias can be used in all the cases discussed below, when the table name is used.

Linking Form Fields to Table Columns

There are two ways to link a form field to a table column, but both of them require the compile-time database to be specified after the DATABASE keyword.

The first method influences only the data type of the form field. It still remains a formonly field, but like with the DEFINE...LIKE statement, its data type is declared LIKE the data type of the specified column. All the rest of the field declaration, including attributes and the formonly prefix remain the same as described in the formonly fields [ATTRIBUTES section](#). Here is its syntax:

```
ATTRIBUTES
field_tag=formonly.field_name TYPE LIKE table.column, [attribute...];
```

This way, the name of the form field will be different from the name of the table column it is linked to. The NOT NULL keywords cannot be assigned to such a field, because whether the field can or cannot accept NULL values depends on whether the column has the NOT NULL constraint or not. However, any other attributes of the column are not applied to the field as well as there is no automatic values verification.

The other method is simpler, but it causes the form field to have the same name as the table column it is linked to:

```
ATTRIBUTES
field_tag=table.column, [attribute...];
```

In this case you do not need to specify the TYPE, you also cannot specify the NOT NULL keywords either for the same reason as in the previous case. If you need to use the field with its prefix, you must use the table name instead of the formonly keyword.



A form field linked to a table column in such a way acquires all the attributes peculiar to the column it is linked to. After 4GL extracts any default attributes and identifies data types from the system catalogue, the association between fields and database columns is broken, and the form cannot distinguish the name or synonym of a table or view from the name of a screen record. Thus you can then treat fields linked to columns in one table as members of one screen record. The screen records which are created automatically in this way are similar to the formonly record. They also do not need to be declared explicitly.

Default Screen Records

Fields which belong to a formonly record cannot belong to any table record. The same is also true for the fields belonging to table records. The membership in an automatically created screen record depends on the prefix used before the field name. Thus a field cannot belong to the formonly record and be linked to a table at the same time, while no field can belong to two or more automatically created records at a time. However, you can combine fields belonging to different default records in an explicitly declared record in the INSTRUCTIONS section.

Below is an example of form fields declaration:

```
DATABASE my_db
SCREEN{
    [f001      ]  [f002      ]
    [f003      ]  [f004      ]
    [f005      ]
}
TABLES
my_table1 my_table2
ATTRIBUTES
    f001=formonly.f001 TYPE LIKE my_table1.item_id, REQUIRED,
    COLOR = REVERSED;
    f002=my_table1.item_name;
    f003=my_table1.date, CENTURY="p";
    f004=my_table2.item_desc;
    f005=my_table2.order_no;
INSRTUCTIONS
    SCREEN RECORD my_rec (formonly.f001, my_table1.item_name,
                           my_table2.order_no)
```

In the example above, there are five fields. Though fields with tags f001, f002 and f003 are linked to the same table, only fields f002 and f003 belong to the same screen record called "my_table1", because specifying the TYPE LIKE keywords does not influence the prefix before the field name which remains formonly. Thus field f001 is the only field which belongs to the formonly screen record.

All in all there are four screen records in the example above one of which is declared explicitly. The first is the formonly record including only one field. Then there is "my_table1" record including two fields, "my_table2" record which also includes two fields, and "my_rec" record which includes three fields belonging to different automatically created records.

Additional form Field Attributes

There are some field attributes which have not yet been discussed. Some of them apply only to form fields linked to table columns, others can be applied to any field.



The VALIDATE LIKE Attribute

If a field has the VALIDATE LIKE attribute, 4GL checks the data in this field against the specified table column using the validation rules that the upscol utility assigned to the specified database column in the syscolval table. It has the same function as the VALIDATE statement and ensures that prohibited data are not inserted into a table.

The syntax of this attribute is as follows:

```
VALIDATE LIKE [table.]column
```

The "table" prefix is the name of an existing table which contains the columns against which the value is validated. You can omit the table prefix, if the column name is unique among the other tables used. In either way the table should be specified in the TABLES section of the form file. The "column" identifier is the name of the column against which the value is verified.

If the field is declared as field_tag=table.column, the VALIDATE LIKE attribute is not required, as such field declaration implies such verification. However, if the field is declared as a formonly field with TYPE LIKE table.column specification or without it, you must include the VALIDATE LIKE attribute, if you want to validate the entered value. Even if all of the form fields are formonly, the form file containing this attribute still requires the database being specified and the TABLES section being included.

Here is an example of the VALIDATE attribute:

```
ATTRIBUTES
f001=formonly.f001 TYPE INT, VALIDATE LIKE ords.item_id;
```

The DISPLAY LIKE Attribute

The DISPLAY LIKE attribute applies all the attributes of a table column to a form field. The attributes for a column retrieved by this attribute are specified in the syscolatt table. Here is its syntax:

```
DISPLAY LIKE [table.]column
```

Like with the VALIDATE LIKE attribute, the table prefix is optional and is required only if the column name is not unique. You also need the database to be specified for the form file and the corresponding table should be included into the TABLES section.

If the field is declared as field_tag=table.column, the DISPLAY LIKE attribute is not required, as such field is displayed with the column attributes by default. However, if the field is declared as a formonly field with TYPE LIKE table.column specification or without it, you must include the DISPLAY LIKE attribute, if you want to the field to use the column attributes. Even if all of the form fields are formonly, the form file containing this attribute still requires the database being specified and the TABLES section being included.

```
ATTRIBUTES
f001=formonly.f001 TYPE LIKE ords.item_id,
DISPLAY LIKE ords.item_id;
```

The attributes are retrieved from the database at compile time, not at runtime. Thus if you change your database, you may need to recompile the application for the new attributes to take effect.

The VERIFY Attribute

The VERIFY attribute does not require the form file to be linked to a database, but it helps to ensure that only correct data are inserted into a table. If this attribute is applied to a field, the user is forced to enter



data in the field twice to reduce the probability of misprints of mistakes during the data input. This attribute is represented by a single VERIFY keyword.

When a user enters the value for the first time and presses RETURN, the cursor is returned to the field once more, the value in it is erased and the user is prompted for one more data entry. If the value entered for the second time differs from the value entered before, an error occurs. This attribute is typically used in INPUT and INPUT ARRAY statements.

```
ATTRIBUTES
f001=formonly.f001 TYPE INT, VERIFY;
f002=tabl1.column2, VERIFY;
```

The Default Values and Attributes of Table Columns

You can change the default values and attributes of table columns by means of the 4GL statements and system tables *syscolval* and *syscolatt* which contain the information about the tables values and attributes, respectively. Both tables are to be created manually and they must strictly correspond the syntax given below.

Setting the Default Volumn Values

The default column values are specified by means of the *syscolva*/table. Each row of this table contains information about the default value:

```
# the SYSCOLVAL table
CREATE TABLE syscolval
(
    tabname nchar(18),          -- contains the name of a table for the column of
                               -- which values are specified
    colname nchar(18),          -- contains the name of a column of which values
                               -- are specified
    attrname nchar(10),          -- contains the type of the value (either
                               -- INCLUDE - for the acceptable value, or
                               -- DEFAULT - for the default value)
    attrval nchar(64)           -- contains the value to be assigned
)

CREATE index sysvcomp on syscolval (tabname,colname)
CREATE index sysvtabname on syscolval (tabname)
```

The indexes are necessary for the table to be used properly by the program. However, the creation of indexes is below the scope of this manual.

The *tabname* column should get the name of a table for the column of which you want to specify values.

The *colname* column should get the name of the column for which you specify values.

The *attrname* column should get the type of the value (either INCLUDE for the acceptable value, or DEFAULT - for the default value).

The *attrval* column should get the value which is to be assigned.

To add default and possible values for one or several columns to this table, you can use any way of database population we have already discussed. For example, you have a table specified as:

```
CREATE TABLE clients (
```



```
country CHAR(3),  
fname CHAR(10),  
lname CHAR(20),  
gender CHAR(6)  
)
```

After you create a *syscolval* table as it is given above, you can use the **INSERT** statement to define possible values for the columns *country* and *gender*:

```
#Specifying three possible values for the column country  
INSERT INTO syscolval VALUES ("clients", "country", "INCLUDE",  
"840")  
INSERT INTO syscolval VALUES ("clients", "country", "INCLUDE",  
"826")  
INSERT INTO syscolval VALUES ("clients", "country", "INCLUDE",  
"124")  
  
#Specifying two possible values for the column gender  
INSERT INTO syscolval VALUES ("clients", "gender", "INCLUDE",  
"male")  
INSERT INTO syscolval VALUES ("clients", "gender", "INCLUDE",  
"female")
```

Therefore, the *country* column can take one of the three values - 840 for the USA, 826 for Great Britain, and 124 for Canada. The *gender* column can take only two values - "male" and "female".

To check how the new values are applied to the *clients* table, let's add some more lines to the source code.

```
WHENEVER ANY ERROR CONTINUE  
LET cntry = "840"  
LET gender = "female"  
{1}VALIDATE cntry, gender LIKE clients.country  
, clients.gender  
IF STATUS = 0 THEN  
    DISPLAY "Everything is all right" at 2,2  
ELSE  
    DISPLAY "Values are invalid" at 2,2  
END IF  
  
LET cntry2 = "105"  
LET gender2 = "child"  
{2}VALIDATE cntry2, gender2 LIKE clients.country,  
clients.gender  
IF STATUS = 0 THEN  
    DISPLAY "Everything is all right" at 3,2  
ELSE  
    DISPLAY "Values invalid" at 3,2  
END IF
```



Here, *ctry* is a variable of CHAR (3) data type, and *gender* is a variable of CHAR(6) data type. After the {1}VALIDATE statement is executed, the *status* variable will get the value 0, because the values of both *ctry* and *gender* variables are within the scope of values possible for the columns *country* and *gender*. After the {2}VALIDATE statement is executed, the *status* variable will get an error code (-1321), because none of the validated variables has a value, available within the corresponding column. The WHENEVER ERROR CONTINUE statement is necessary because, if it is not specified, the {2}VALIDATE statement will cause the program termination.

As the result of the program execution, the following will be displayed to the screen:

```
Everything is all right
Values are invalid
```

You can specify default values for the table columns the following way:

```
INSERT INTO syscolval VALUES ("clients", "country", "DEFAULT",
"124")

INSERT INTO syscolval VALUES ("clients", "gender", "DEFAULT",
"male")
```

To ascertain that the *country* and *gender* columns have default values, you can use the INITIALIZE statement:

```
INITIALIZE ctry, gender LIKE clients.country, clients.gender
DISPLAY "The default value of the country column is ", ctry AT 4,2
DISPLAY "The default value of the gender column is ", gender AT 5,2
```

These lines will display the default values of the columns *country* and *gender*.

Setting Column Attributes

The second table you can create and use to modify database settings is the *syscolatt* table. This table specifies table attributes, such as colour, font, format, etc. As the *syscolval* table, the *syscolatt* table should have an algoristic structure and indexes, which are as follows:

```
CREATE TABLE "informix".syscolatt
(
    tabname nchar(18),
    colname nchar(18),
    seqno serial not null ,
    color smallint,
    inverse nchar(1),
    underline nchar(1),
    blink nchar(1),
    left nchar(1),
    def_format nchar(64),
    condition nchar(64)
);
REVOKE ALL ON "informix".syscolatt FROM "public";

CREATE INDEX "informix".sysacomp ON "informix".syscolatt (tabname, colname,
seqno);
CREATE INDEX "informix".sysatabname ON "informix".syscolatt (tabname);
```

The columns in the *syscolatt* table should get the following values:



- *Tabname* column gets the name of the table where the modified column is situated;
- *Colname* column gets the name of the column which is to get new attributes;
- *Seqno* column gets the serial number
- *Color* column gets an integer number specifying the display attributes of the text. The *color* column can get four values: "1" for Normal, "2" for Dim, "3" for Bold, and "4" for Invisible.
- *Inverse* column should get a character "y" or "n", where "y" means that the value should be displayed in reverse, and "n" means that the value should be displayed in normal mode.
- *Blink* column should get a character "y" or "n", where "y" means that the value should blink when displayed, and "n" means that the value should be displayed in normal mode.
- *Left* column should get a character "y" or "n", where "y" means that the value should be left-aligned when displayed to a form field, and "n" means that the alignment will be according to the default the data type display settings.
- *Def_format* column should contain the default display format for the values stored in the corresponding column. The format should be specified as in the USING operator.
- *Condition* column contains the condition which should be satisfied in order for the attributes to be applied to the value stored in the corresponding column.

Example

This example illustrates how you can link variables and form fields to specific database tables and columns.

Here is the source code file:

```
#####
# This application shows how the views and synonyms are created.
# It also shows how the form fields are linked to database tables.
#####
DATABASE db_examples

DEFINE
  # This variable has the same data type as id_country column, as this
  # column has SERIAL data type, the variable defined is of the INT data type

    v_id_country LIKE clients.id_country,

  # variable v_gender will be of the same data type as clients column, that
  # is of CHAR(6)
    v_gender      LIKE clients.gender,

  # record r_clients will have as many members as clients table has columns
  # they will have matching names and data types and will be specified in the
  # same order
    r_clients     RECORD LIKE clients.*,
      num_key      INTEGER,
      country_name_var VARCHAR (20),
      err INTEGER,
      age INTERVAL YEAR TO MONTH,
      age_y INTERVAL YEAR TO YEAR

MAIN
OPTIONS FORM LINE 1
```



```
MENU "Main"
    COMMAND "Views and Synonyms"
        "Create different views and synonyms"

        MENU "Views and Synonyms"
            COMMAND "Create"
                "Creates views and synonyms"
                CALL views()

            COMMAND "Drop"
                "Drops the created views and synonyms"
                CALL drop_views()
            COMMAND "Return"
                "Returns to the previous menu"
                EXIT MENU
        END MENU

    COMMAND "Validate"
        "We use the VALIDATE statement to verify the values assigned."
        CALL scr2_validate()

    COMMAND "Initialize"
        "The INITIALIZE statement is used to assign the default values"
        CALL scr3_initialize()

    COMMAND "Input Form"
        "Demonstrates how the input can be performed using the database data"
        CALL scr4_form()

    COMMAND "Exit" "Exit the demo program"
        CALL drop_views() -- we need to clean the database before exiting
        EXIT PROGRAM

END MENU
END MAIN

#####
# This function creates a set of database table views and synonyms
#####

FUNCTION views()
    # Like with the creation of the tables, we need to ensure that no error
    # occurs even if there is no view that could be deleted.

    WHENEVER ERROR CONTINUE

    # The view created below contains all the columns of table clients and all
    # its rows
    CREATE VIEW clients_view_1 AS SELECT * FROM clients

    #The err variable is used to check whether any error occurred. If the view
    # to be created already exists, the variable will get the value -310 and
    # the function execution will be terminated
```



```
LET err = sqlca.sqlcode
IF err = -310 THEN
    CALL fgl_winmessage( "Attention",
        "You have already created the synonyms and views",
        "Error")
    EXIT FUNCTION
END IF

# The view created below contains some of the columns from clients and all
# its rows. Of course, the rows do not include the data from the omitted
# columns
CREATE VIEW clients_view_2 (id_client, id_country, first_name, last_name,
date_birth, gender, address, city, start_date, end_date)
AS SELECT id_client, id_country, first_name, last_name, date_birth, gender,
address, city, start_date, end_date
FROM clients

# The view below contains some of the columns of clients table and a limited
# set of rows
CREATE VIEW clients_view_3
(id_client_v,first_name_v,last_name_v,start_date_v,end_date_v)
AS SELECT id_client,first_name,last_name,start_date,end_date
FROM clients
    WHERE id_client BETWEEN 3 AND 5
    WITH CHECK OPTION

# Then we create the synonyms for the tables
CREATE SYNONYM clients_of_bank FOR clients
CREATE PUBLIC SYNONYM all_country FOR country_list
CREATE SYNONYM my_clients FOR clients_view_2

# Te message informs the user that the views and synonyms have been
# created successfully
CALL fgl_winmessage( "Done!",
    "Several views and synonyms have been created" ||
    "\nYou can view the structure of the views and synonyms" ||
    "\n using an external tool." ||
    "\n E.g. if you run the application against Informix-server" ||
    "\n use the dbschema utility.",
    "Info")
END FUNCTION

#####
# This function is used to drop the views and synonyms created before
#####

FUNCTION drop_views()
WHENEVER ERROR CONTINUE
    DROP VIEW clients_view_1 -- we delete views, if they exist already

# The variable err gets the value -206 if there is no view to be dropped
# In this case, the function execution terminates

    LET err = sqlca.sqlcode
    IF err = -206 THEN
```



```
CALL fgl_winmessage( "Attention",
    "There are no synonyms or views to delete",
    "Error" )
EXIT FUNCTION
END IF

DROP VIEW clients_view_2
DROP VIEW clients_view_3
WHENEVER ERROR STOP

# Then again use the WHENEVER statement to prevent the program from
# terminating, if the synonyms do not exist yet.
# We need to drop the synonyms beforehand, because of the program is run more
# than once you will get errors otherwise.
WHENEVER ERROR CONTINUE
    DROP SYNONYM clients_of_bank -- the synonym will be dropped if it already
                                  -- exists
    DROP SYNONYM all_country
    DROP SYNONYM my_clients
WHENEVER ERROR STOP

CALL fgl_winmessage( "Done!",
    "Created views were deleted",
    "Info" )

END FUNCTION

#####
# This function is used to demonstrate how the entered values can be
# verified using the VALIDATE statement
#####

FUNCTION scr2_validate()
    # We call the function which creates the syscolval table
    # with the acceptable and default values
    # for columns id_country and gender of table clients
    CALL like_upscol()

    # we assign the values to the variables which are both valid
    # because they are in the syscolval table
    LET v_id_country = "804"
    LET v_gender = "male"

    # We need the WHENEVER again while we need the program to continue even if
    # the value of the variable is verified as out of the acceptable range.
    WHENEVER ERROR CONTINUE
        # We check two variables against two database columns in one VALIDATE
        # statement
    OPEN WINDOW win_1 AT 1,1 WITH FORM "display_form"
        DISPLAY "!" TO b001
        DISPLAY "We use the VALIDATE statement to verify the values assigned"
        TO headr
```



```

VALIDATE v_id_country,v_gender LIKE clients.id_country,clients.gender
DISPLAY "These are valid values:" AT 5,2
DISPLAY "v_id_country = ",v_id_country AT 6,2
DISPLAY "v_gender      = ",v_gender      AT 7,2
DISPLAY "status        = ",status        AT 8,2
                                         -- both are valid, status must be 0
WHENEVER ERROR STOP

LET v_id_country = "826" -- the value is valid
LET v_gender      = "sex"  -- the value is invalid

WHENEVER ERROR CONTINUE
VALIDATE v_id_country,v_gender LIKE clients.id_country,clients.gender
DISPLAY "One of the values is invalid:" AT 10,2
DISPLAY "v_id_country = ",v_id_country AT 11,2
DISPLAY "v_gender      = ",v_gender      AT 12,2 ATTRIBUTE (RED, BOLD)

# As one of the values is invalid, the status gets -1321 value
DISPLAY "status        = ",status        AT 13,2 ATTRIBUTE (YELLOW, BOLD)
WHENEVER ERROR STOP

# This section of the function makes the program react on the button that
# the form contains. It is an alternative to the ON KEY clause during
# an input
LABEL getk_2:
LET num_key = FGL_GETKEY()

IF num_key = 3009 THEN
    CLOSE WINDOW win_1 -- 3009 corresponds to the F3 button
    CLEAR SCREEN
ELSE GOTO getk_2      -- if the user presses any other button, the
                        -- program returns to the fgl_getkey() function
END IF

END FUNCTION

#####
# This function is used to demonstrate how the INITIALIZE statement
# Can be used to assign values to variables
#####
FUNCTION scr3_initialize()
    CLEAR SCREEN
    OPEN WINDOW win_1 AT 3,2 WITH FORM "display_form"
    # We initialize two variables, they get values specified for these columns
    # in the syscolval table
    DISPLAY "!" TO b001
    DISPLAY "The INITIALIZE statement is used" ||
    "to assign the default values to the variables" TO headr

    INITIALIZE v_id_country,v_gender LIKE clients.id_country,clients.gender
    DISPLAY "v_id_country = ",v_id_country," v_gender = ",v_gender AT 7,2

    LABEL getk_3:
    LET num_key = FGL_GETKEY()

```



```
IF num_key = 3009 THEN
    CLOSE WINDOW win_1
    CLEAR SCREEN
ELSE GOTO getk_3
END IF

END FUNCTION

#####
# This function demonstrates how the VALIDATE form field attribute
# can be used to specify the values allowed in the field
#####

FUNCTION scr4_form()

OPEN WINDOW w_clients AT 1,1 WITH FORM "clients" ATTRIBUTE (FORM LINE 2)
DISPLAY "!" TO b001
DISPLAY "!" TO b002

DISPLAY "We use the VALIDATE form attribute to control the values",
        " displayed in the form fields" AT 1,2 ATTRIBUTE(GREEN, BOLD)

OPTIONS
    ACCEPT KEY F3,
    INPUT WRAP

# We retrieve one row from the table with id_client = 1
SELECT c.* INTO r_clients.* FROM clients c WHERE c.id_client = 1;

INPUT r_clients.id_client,
       r_clients.first_name,
       r_clients.last_name,
       r_clients.date_birth,
       r_clients.gender,      -- this value will be validated by the attribute
       r_clients.id_passport,
       r_clients.date_issue,
       r_clients.date_expire,
       r_clients.address,
       r_clients.city,
       r_clients.zip_code,
       r_clients.id_country, -- this value will be validated by the attribute
       r_clients.add_data,
       r_clients.start_date,
       r_clients.end_date
       WITHOUT DEFAULTS
FROM s_clients.*

ON KEY (F10)
    EXIT INPUT

# This AFTER FIELD block is used to calculate the age of the
# client
AFTER FIELD date_birth
    IF r_clients.date_birth IS NOT NULL AND
```



```

        r_clients.date_birth > "12/31/1899"
THEN LET age = EXTEND(TODAY ,YEAR TO MONTH)
      - EXTEND(r_clients.date_birth ,YEAR TO MONTH)
ELSE LET age = NULL
END IF
DISPLAY age USING "yy" TO age

# We use this AFTER FIELD block to find the country name which
# corresponds to the chosen code and to display it to the
# corresponding form field
AFTER FIELD id_country
    SELECT country_name INTO country_name_var FROM country_list
                           WHERE id_country = r_clients.id_country
    DISPLAY country_name_var TO country_name
AFTER INPUT
    CALL fgl_winmessage( "Saved" , "The values were saved" , "info" )
    CONTINUE INPUT

END INPUT
CLOSE WINDOW w_clients
CLEAR SCREEN

END FUNCTION

#####
# like_upscol has the same effect which is achieved by the upscol utility
# for specifying the default and acceptable values
# This system table, if created using 4GL code, must have absolutely identical
# structure with the syscolval table created by automatically by the upscol
# utility
#####
FUNCTION like_upscol()

# We delete and then recreate the system table syscolval
# It is required, if this function was run before
# and the table already exists
WHENEVER ERROR CONTINUE
    DROP TABLE syscolval
WHENEVER ERROR STOP

CREATE TABLE syscolval
    # these are the necessary columns which
    # should be present in this system table
    (
        tablename nchar(18) ,      -- contains the name of a table for the column of
                                -- which values are specified
        colname nchar(18) ,       -- contains the name of a column of which values
                                -- are specified
        attrname nchar(10) ,      -- contains the type of the value (either
                                -- INCLUDE - for the acceptable value, or
                                -- DEFAULT - for the default value)
        attrval nchar(64)        -- contains the value to be assigned
    )
CREATE index sysvcomp on syscolval (tablename,colname)
                                -- the indexes are necessary, because they are

```



```
-- created by the upscol
CREATE index sysvtabname on syscolval (tabname)
-- the creation of indexes below the scope of this
-- manual though

# We specify the acceptable values for the id_country column: 276, 084, 826
INSERT INTO syscolval VALUES("clients","id_country","INCLUDE","276")
INSERT INTO syscolval VALUES("clients","id_country","INCLUDE","084")
INSERT INTO syscolval VALUES("clients","id_country","INCLUDE","826")

# We also specify the acceptable values for gender column: male, female
INSERT INTO syscolval VALUES("clients","gender","INCLUDE","male")
INSERT INTO syscolval VALUES("clients","gender","INCLUDE","female")

# We specify "804" as the default value for the column id_country
INSERT INTO syscolval VALUES("clients","id_country","DEFAULT","804")
# We specify "male" as the default value for the column gender
INSERT INTO syscolval VALUES("clients","gender","DEFAULT","male")

END FUNCTION
```

Form Files

Here is the form stored in the 'display_form.per' form file
DATABASE formonly

```
SCREEN {

\gp-----
\g| [f001] ] | \g
\g| [f001] ] | \g
\g| [b001] ] | \g
\gb-----d\g

}
```

ATTRIBUTES

```
f001 = formonly.headr, widget = "label", wordwrap,
COLOR = Green bold;

b001 = formonly.b001,
config = "F10 {Back}", widget = "button",
```



```
COMMENTS = "Go back to the main menu";
```

Below is given the form to where the input will be performed (the 'clients.per' file)

```
DATABASE db_examples
SCREEN SIZE 26 BY 80
```

```
{
\gp-----[101]-----q\g
\g|\g      ID:\g[f00      ]
\g|\g      First Name:\g[f01      ]
\g|\g      Last Name:\g[f02      ]
\g|\g      Birth Date:\g[f03      ]\g Age:\g[f15      ]
|\g
\g|\g      Gender:\g[f04      ]
\g|-----| \g
\g|\g      ID_Passport:\g[f05      ]\g Issued on:\g[f06      ]\g Expires on:\g[f07
] |\g
\g|-----| \g
\g|\gAddress:\g[f08      ]
\g|\g      City:\g[f09      ]
\g|\g      Zip:\g[f10      ]
\g|\g      Country:\g[f11      ][f16
\g|-----| \g
\g|[f12      ]
\g|[f12      ]
\g|[f12      ]
\g|-----| \g
\g|\g      Registered:\g[f13      ]\g Deregistered:\g[f14      ]
|\g
\g|-----| \g
\g|[b001      ]\g [b002      ]\g
\gb-----d\g
}
```

```
TABLES clients
```

ATTRIBUTES

```
f15 = FORMONLY.age;
f16 = FORMONLY.country_name TYPE CHAR;

f00 = FORMONLY.id_client      TYPE LIKE clients.id_client,NOENTRY;
f01 = FORMONLY.first_name     TYPE LIKE clients.first_name,REQUIRED,
      COMMENTS=" Enter the first name";
f02 = FORMONLY.last_name      TYPE LIKE clients.last_name,REQUIRED,
      COMMENTS=" Enter the last name";
f03 = clients.date_birth,    FORMAT="mm.dd.yyyy",PICTURE="#.#.##.###",
      COMMENTS=" Enter the date birth";
f04 = clients.gender,        VALIDATE LIKE clients.gender,
      -- the values allowed for entering are only those
      -- specified in the syscolval table for the gender column
      --widget = "combo", include = ("male","female"),
      REVERSE,COMMENTS="Chose the gender (male, female)";
f05 = clients.id_passport,   VERIFY,COMMENTS=" Enter the passport number";
f06 = clients.date_issue,    FORMAT="mm.dd.yyyy",PICTURE="#.#.##.###",
      COMMENTS=" Enter the date the passport was issued";
f07 = clients.date_expire,   FORMAT="mm.dd.yyyy",PICTURE="#.#.##.###",
```



```
    COMMENTS=" Enter the date when passport expires";
f08 = clients.address,      REQUIRED,
      COMMENTS=" Enter the street address, apartament,housing, etc.";
f09 = clients.city,        COMMENTS=" Enter the city";
f10 = clients.zip_code,    COMMENTS=" Enter the zip code for this address";
f11 = clients.id_country,  VALIDATE LIKE clients.id_country,
                           -- the values allowed for entering are only those
                           -- specified in the syscolval table for the id_country column
                           REVERSE, REQUIRED,
                           COMMENTS=" Chose the county ID";
f12 = clients.add_data,    WORDWRAP,COMMENTS=" Enter the additional data";
f13 = clients.start_date,  DEFAULT = TODAY,
                           FORMAT="mm.dd.yyyy", PICTURE="#.#.##.####", REQUIRED,
                           COMMENTS=" Enter the date of registered";
f14 = clients.end_date,   FORMAT="mm.dd.yyyy", PICTURE="#.#.##.####", REQUIRED,
                           COMMENTS=" Enter the date of deregistered";

b001 = formonly.b001,
       config = "F3 {Save}", widget = "button",
       COMMENTS = "Save the inutted values";

b002 = formonly.b002, config = "F10 {Back}", widget = "button",
       COMMENTS = "Go back to the main menu";

101 = formonly.headr, widget = "Label", config = "Clients data",
      color = blue bold;

INSTRUCTIONS
  DELIMITERS "[ ]"
  SCREEN RECORD s_clients (id_client THRU end_date)
```



The Notion of Cursor

For now we have only discussed the way to select one row from a table. Though some statements as CREATE VIEW or UNLOAD have the SELECT clause which is able to select more than one row, you need to use a cursor to retrieve the values of multiple rows into program variables and to process them. A cursor is also useful in the process of inserting multiple rows into a table or deleting them from it.

Consequently there are several types of cursors:

- Cursor, associated with a SELECT statement which can be used for selecting, updating, or deleting multiple rows. It is called a select cursor or an update cursor.
- Cursor, associated with a restricted INSERT statement

This chapter will dwell upon the cursor associated with the SELECT statement.

Declaring a Read-Only Cursor

A read-only cursor is used to retrieve multiple rows from a database. It cannot be used for modifying a database in any way.

A program that retrieves multiple rows should follow such pattern:

- The cursor should be declared and associated with the SELECT statement.
- Then the cursor should be opened, and the execution of the associated SELECT statement begins.
- The program fetches a row of data into host variables at a time and processes it.
- After the last row has been processed, the program closes the cursor.
- You should free the cursor, if it is no longer needed.

Use the DECLARE statement to declare a cursor. This statement gives the cursor a name, specifies its type, and associates it with a statement. In our case, it is associated with the SELECT statement. The syntax of the DECLARE statement for the insert and for the select cursors differ. The general syntax of the read-only cursor declaration associated with the SELECT statement is as follows:

```
DECLARE cursor_name CURSOR FOR SELECT_statement
```

The cursor declared in such a way can be used only to retrieve multiple rows from a database, it cannot be used to change the database data in any way. There are other select cursor subtypes able to change the contents of the table, but they will be discussed later in this chapter.

The cursor name in the pattern above can be either a variable of a character data type containing the name, or the cursor identifier itself (which should not be enclosed in quotation marks). The cursor name must be unique among other cursors declared within the program.

When the cursor is used with a SELECT statement, it is a data structure that represents a specific location within the active set of rows that the SELECT statement retrieved. A cursor has the global scope of reference, thus once declared in any program module, it can be used from within any other module.



The CURSOR Data Type

Querix4GL offers another way of working with cursors by introducing the CURSOR data type. A CURSOR data type variable is declared in the following way:

```
DEFINE variable_name CURSOR
```

Here, *variable_name* stands for an identifier which will be later referenced by various methods in order to manipulate the cursor. The word CURSOR designates the data type for the variable. For example:

```
DEFINE cur CURSOR
```

The declaration above actually creates an object that describes the cursor. Then one can call on that object a number of functions (or methods) necessary to open, set parameters, specify an SQL query, etc. These methods are described in the present and the next chapters.

	Note: A variable of the CURSOR data type can be declared of any scope of reference.
--	--

The SELECT Statement Associated with a Cursor

The syntax of the SELECT statement associated with a cursor is the same as of a stand-alone SELECT statement:

```
SELECT select_list [INTO clause] FROM table_list
                  [OUTER table_list] [WHERE clause]
```

As you can see, the INTO clause is optional here. You can select whether you want to specify the variables to retrieve the values here, or in the FETCH statement which actually retrieves the rows from the tables. The FETCH statement is discussed later in this chapter. The WHERE statement restricts the number of the selected rows.

The rows selected by the SELECT statement associated with the cursor are called the active set. The cursor is allowed to move only among the rows which constitute the active set and cannot go beyond it. If the WHERE clause is absent, all the rows of the table specified belong to the active set.

Here are two examples of the DECLARE statement one of which contains the INTO sub-clause whereas the other does not:

```
DECLARE my_curs1 CURSOR FOR
SELECT * INTO my_rec.* FROM my_tab

DECLARE my_curs2 CURSOR FOR
SELECT col1, col2, col3 FROM my_tab1 WHERE col1 > 1000
```



The SELECT Statement Associated with a Variable of the CURSOR Data Type

If you defined a variable of the CURSOR data type, you must call the *Declare()* method in order to specify an SQL statement this cursor will execute, and determine the type to which it belongs. The syntax used to call the method is as follows:

```
CALL variable_name.Declare ("SQL_statement")
```

Declare() can take three parameters. However, only the first parameter is obligatory. The optional parameters will be discussed in the following chapters. The first and the only parameter required by the method is an SQL statement which can be either a quoted string representing a standard SQL statement, or a variable of a character data type which stores this statement. Here we deal only with SELECT, although it can also be another type of SQL statement.

```
"SELECT * FROM tab_1"  
OR  
DEFINE quer CHAR (40)  
LET quer = "SELECT * FROM tab_1"
```

The OUTER Keyword

Two or more tables can be joined in order to perform queries on the data they contain. This can be done by using simple and outer joins.

The result of a simple join is a set of rows which meet the conditions specified in the WHERE clause of the SELECT statement.

```
SELECT * FROM tab_1, tab_2 WHERE tab_1.col_1 = tab_2.col_1
```

The above query will return the rows from both tables in which the values in col_1 of table tab_1 are equal to those in col_1 of table tab_2, and a single temporary table with all the retrieved values will be created. The rows that don't meet the select criteria are dropped. The WHERE statement must include a special condition binding the tables together which is called join condition of the following type: tab1.col1=tab2.col2.

An outer join, in contrast to a simple join where the tables to be joined are considered equal, treats one table as a dominant one and the other or others as subservient. To change a simple join into an outer join, add the keyword OUTER before the name of the subservient table. The join criteria must be specified in the WHERE clause.

There are several types of outer join, but we will describe only two of them, which are more commonly used:

1. A simple outer join on two tables
2. A simple outer join to a third table

With a simple outer join on two tables the result of a query will include all the rows that satisfy the conditions in the WHERE clause of the SELECT statement but will not be restricted by them. The rows from the dominant table that don't satisfy those conditions are not discarded as in the case of a simple join, even though there are no matching rows in the subservient table. In general the rows are retrieved from the dominant table using the following rules:

- If the WHERE clause contains only the join conditions, all the rows will be selected from the first table (tab1), which is dominant:



```
SELECT * FROM tab1 OUTER tab2 WHERE tab1.col1=tab2.col1
```

- If the WHERE statement has a separate condition applying only to the first table, the rows retrieved from the first table will be specified by this condition (which is tab1.col2>100 in our case).

```
SELECT * FROM tab1 OUTER tab2  
WHERE tab1.col1=tab2.col1 AND tab1.col2>100
```

However, only those rows which satisfy the join condition are retrieved from the second table, which is the subservient one. Consequently, the rows of the dominant table which do not have a matching row from the subservient table will be followed by NULL values instead.

```
SELECT * FROM tab_1 OUTER tab_2 WHERE tab_1.id = tab_2.id
```

This query will return all the rows from table tab_1 and only those rows from tab_2 which meet the select condition.

An outer join to a third table requires a simple join on two subservient tables to be executed first. Then the outer join is performed in order to combine the result of the simple join with the information from the dominant table. The keyword OUTER must be followed by the names of the subservient tables enclosed in parentheses and separated by commas.

In the query below, tables tab_2 and tab_3 are first joined by means of a simple join (only the rows meeting the join condition are retrieved), after which the information returned as a result is combined with the data from tab_1. The rows in tab_1 without matches in the subservient tables will be followed by NULL values in the same way as it happens with a simple outer join.

```
SELECT * FROM tab_1 OUTER (tab_2, tab_3)  
WHERE tab_1.id=tab_2.id AND tab_2.id=tab_3.id
```

	Note: You need to specify the join condition for all the three tables.
---	---

Opening a Cursor

The DECLARE statement is not an active statement. In the same way as the DEFINE statement declares variables and allocated memory for them, but does not assign any values, the DECLARE statement declares a cursor and allocates storage for it, but you need another statements to actually use that cursor.

When you are ready to use the cursor which has been declared, you should specify it in the OPEN statement. A cursor that was not opened cannot be used in any statements other than the OPEN statement. The OPEN statement has a very simple syntax:

```
OPEN cursor_name
```



When 4GL encounters this statement, it checks whether the cursor specified in it was declared earlier and then passes the SELECT statement associated with the cursor to the database. The search for the matching rows begins. Then after the first row matching the search criteria is found, it is not actually returned. To return a row, you need to use the FETCH statement described below.

Here is an example of a cursor being declared and then opened:

```
DECLARE my_curs1 CURSOR FOR
SELECT * INTO my_rec.* FROM my_tab
OPEN my_curs1
```

When the cursor is opened, **sqlcode** element of the built-in record **sqlca** is set to the value of the code returned by the OPEN statement. While *sqlca.sqlcode*=0, the SELECT statement is valid and the cursor can be used. It can be set to negative values, if some errors are present in the SELECT syntax or the database is inaccessible for some reason.

Opening a Cursor Associated with a Variable of the CURSOR Data Type

To open the cursor associated with a CURSOR variable you declared previously, you should call the *Open()* method. It requires no parameters, and is the equivalent of the standard OPEN statement.

```
CALL variable_name.Open()
```

This line of code given below opens a cursor associated with the CURSOR variable named *cur*.

```
CALL cur.Open()
```

The *Open()* method returns a value of the *sqlca.sqlcode* global record member. If the method executes without errors, this value is set to 0. Otherwise, a negative integer representing the error code is returned.

Fetching the Rows

To process the rows the FETCH statement is used. It moves the cursor through the rows which are determined by the SELECT statement associated with the cursor and fetches them one at a time. The cursor cannot be moved to the rows of a table which do not belong to the active set. The FETCH statement can be used with a cursor which has already been opened and declared. Here is its syntax:

```
FETCH cursor_name [ INTO clause ]
```

Each time 4GL executes the FETCH statement it retrieves and processes a single row, so to retrieve multiple rows, it needs to be processed a number of times. This statement is typically used in a conditional loop, which means in a WHILE or FOR statement, so that each row it retrieves gets displayed or is processed in any other way.

As a rule, the FETCH statement is used in a WHILE loop which condition tests the value of the *sqlca.sqlcode* built-in variable. In such a way the rows are retrieved from the table until there are no rows left which fit the select condition or until the cursor becomes unavailable due to other reasons.



The INTO Clause

The INTO clause of the FETCH statement is optional. However, there must be exactly one INTO clause in the whole construction which selects and retrieves the rows. It must be either in the SELECT statement associated with the cursor, or in the FETCH statement processing that cursor. You cannot include the INTO clause in both of them.

Here is an example of a FETCH statement without the INTO clause. The WHILE statement tests the value of the built-in variable and while the there is no errors it processes the FETCH statement.:.

```
DECLARE c_item_id CURSOR FOR
    SELECT item_id, item_price, am_left
    INTO i_id, i_price, in_stock
    FROM items
OPEN c_item_id
WHILE sqlca.sqlcode = 0
FETCH c_item_id
IF SQLCA.SQLCODE = 0 THEN
DISPLAY o_num, i_num, s_num AT 2,2
SLEEP 1
END IF
END WHILE
```

Fetching the Rows with a CURSOR Variable

After a cursor associated with a CURSOR variable has been declared, you can determine the variables which will store the results retrieved by it. This is done with the help of the *SetResults()* method which can be regarded as the equivalent of the INTO clause in the FETCH statement. The parameter of the method is the list of comma-separated variables which will receive values. The length of the list can vary depending on the number of values to be returned. *SetResults()* can be called at any point in the cursor life cycle.

```
CALL variable_name.SetResults(var1, [var2], .. [varN])
```

Here is an example of *SetResults()* being called.

```
DEFINE cur CURSOR,
id INT, first_name CHAR (20) -- Variables to store the
-- values returned by the cursor

CALL cur.Declare("SELECT * FROM tab_1")
CALL cur.SetResults(id, first_name) -- The SetResults() method will
-- send the values returned by
-- the cursor to the variables id
-- and firstname
...
```

Although in the code above the *setResults()* method follows *Declare()*, it can actually precede it, for, as has already been said, it can be located at any point in the cursor life cycle.

Once you have set the variables to store the results of the cursor operation, you can call the *FetchNext()* method in order to retrieve rows from the active set.

```
CALL variable_name.FetchNext()
```

This method takes no parameters, retrieves the row which is next to the current one and records the values into the variables specified by the *SetResults()*



method. *FetchNext()* returns a value of *sqlca.sqlcode*. If the value is 0, it means that some rows still remain in the active set. Once the cursor attempts to retrieve a row beyond the current active set, the value of *sqlca.sqlcode* is set to 100 (the NOT FOUND condition). If some errors occur, a negative number representing the error code is returned.

The method is typically used within a conditional loop, most often WHILE.

For example:

```
...
DEFINE cur CURSOR, i INT, f_name CHAR (30)

CALL cur.Declare ("SELECT * FROM tab_1")

CALL cur.Open ()

CALL cur.SetResults(i, f_name)

WHILE (cur.FetchNext() = 0) -- The loop will continue until the value of
-- sqlca.sqlcode is set a non-zero value, that
-- is until an empty row is fetched,
-- or an error occurs

DISPLAY i, f_name

END WHILE

...
```

The Sequence of Fetching Rows

The cursor is not something you can see, it is not like the screen cursor. After it is declared, it moves about the database table. Then each time the FETCH statement is executed, the current position of the cursor is moved to the row which was retrieved last and remains there until another FETCH statement is executed and it is moved again.

The order in which the cursor moves around the table depends on the cursor type. There are cursors which can retrieve rows from anywhere within the table and those which can retrieve rows only one by one. The type of the cursor we discuss in this chapter is the sequential one which retrieves the rows according to the order in which they are stored in the database. Thus the rows are fetched one by one in the sequence in which they are located in the table. So each time the FETCH statement is executed, it fetches the row next to the one where the cursor currently resides.

Because the sequential cursor can retrieve only the next row, the database server can frequently create the active set one row at a time. On each FETCH operation, the database server returns the contents of the current row and locates the next row.



Returning Information about the Cursor

Querix 4GL supports a function that returns the current cursor name. This is the *cursor_name()* function. The function takes one parameter which is the quoted identifier of the cursor, the name of which you want to return. The cursor name is generated automatically and consists of the word *Cursor* and the number starting from 0. For example: *Cursor0* or *Cursor1*.

In the following example the variable *curname* will get the name associated with the cursor specified as *cur*:

```
DATABASE s_auth
MAIN
DEFINE rec RECORD
    cntry CHAR (3),
    fname CHAR (10),
    lname CHAR (20),
    gen CHAR (6)
END RECORD,
i INT,
curname CHAR (18)

DECLARE cur CURSOR FOR
    SELECT * INTO rec.* FROM clients
OPEN cur
LET curname = cursor_name("cur")-- returns Cursor0
DISPLAY curname AT 5,2
CALL fgl_getkey()
END MAIN
```

Getting the Cursor Name for a Variable of the CURSOR Data Type

If a cursor is associated with a variable of the CURSOR data type, you can get information about it by calling the *GetName()* method.

```
CALL variable_name.GetName()
```

variable_name here, as usual, stands for the variable of the CURSOR data type declared previously. As can be seen, the method takes no parameters and returns a character string, which is the internal name given to the cursor by the compiler. This is a mingled name which consists of the word *Cursor* and the number starting from 0. For example: *Cursor0* or *Cursor1*.

In the following example the variable *cur_name* will store the name of the cursor associated with a variable of the CURSOR data type.

```
DEFINE cur CURSOR, cur_name CHAR (20)
CALL cur.Declare ("SELECT * FROM tab_1")
CALL cur.Open()
...
LET cur_name = cur.GetName()
DISPLAY cur_name AT 2, 2
...
```

Note that you can determine the internal name of the cursor by means of the *GetName()* method even after a cursor was closed. However, it will not work if you have freed the resources allocated to a cursor. The *Close()* and *Free()* methods used for these purposes are discussed next.



Closing and Freeing the Cursor

After you have retrieved all the rows, you should close the cursor, if you do not plan to use it in another `FETCH` statement. You can also free it afterwards to release the memory allocated to it.

The CLOSE Statement

To close the cursor, use the following statement:

```
CLOSE cursor_name
```

After the cursor is closed, you can no longer reference it with any other statements than `OPEN` and `FREE`. If you try to close a cursor which has not been opened or which has already been closed once, no action will be taken by 4GL. When the cursor associated with a `SELECT` statement is closed, that `SELECT` statement is terminated.

Closing a Cursor Declared by Means of a CURSOR Variable

To close a cursor associated with a `CURSOR` data type variable you should call the `Close()` method. It doesn't take any parameters, and returns a value of `sqlca.sqlcode`. For example, the following statement closes the cursor associated with variable `cur` we used in the previous examples:

```
CALL cur.Close()
```

Note that after closing a cursor you can reference it only by two methods `Free()` and `Open()`. Also, you can still find out its internal name by calling the `GetName()` method since closing a cursor doesn't remove it from the memory.

The FREE Statement

The `FREE` statement releases resources that are allocated to a cursor. Its syntax is as follows:

```
FREE cursor_name
```

The amount of available memory in the system limits the total number of open cursors that are allowed at one time in one process. Use the `FREE` statement to release resources that a cursor holds. Whereas you can reopen a closed cursor and use it again, once freed, the cursor cannot be used by any statements. To use it again you need to declare it once more. You cannot free a cursor which has not been explicitly closed.

Releasing Resources Allocated to a Cursor Declared Through a CURSOR Variable

To free the resources allocated to a cursor which was declared with the help of a `CURSOR` variable, the `Free()` method should be called. It requires no arguments, and returns a value of `sqlca.sqlcode`.

```
CALL cur.Free()
```

After this statement executes, you will not be able to open and use the cursor again.



	<p>Note: The <i>Free()</i> method is implicitly called when all references to the cursor object are closed.</p>
---	--

Example

This example operates a number of read-only cursors which retrieve information from a single table as well as from a number of tables at once.

For this application to work you need to be connected to the database specified in the DATABASE statement. Your database must also contain the tables which can be created by [Databases in 4GL](#) example and populated by [Populating Database Tables](#) example before you can run this application.

```
#####
# This example illustrates the use of different types of cursors to retrieve
# data from database tables
#####

DATABASE db_examples

MAIN
DEFINE
    r_clients RECORD LIKE clients.*,
    r_contracts    RECORD LIKE contracts.*,
    r_country_list RECORD LIKE country_list.*,
    i              INT,
    cur_name       CHAR ( 7 ),                      -- This variable will store
                                                       -- names the program gives to
                                                       -- cursors
    curs_name      CHAR ( 80 ),                     -- We will assign a cursor name
                                                       -- to this variable and use it
                                                       -- to reference the cursor
    c_cur_5        CURSOR                          -- A variable of the CURSOR data
                                                       -- type

MENU "Cursor"

COMMAND "All Records" "Selecting all the rows from one table"
OPEN WINDOW cur_window AT 2,2 WITH form "cur_form"
# We declare c_cur_1 cursor associated with a SELECT statement
# which retrieves all the data from the clients table
DECLARE c_cur_1 CURSOR FOR
    SELECT * FROM clients

OPEN c_cur_1 -- we open the cursor

LET cur_name = cursor_name("c_cur_1") -- We use the cursor_name()
                                         -- function to determine
                                         -- the database internal name of
                                         -- the cursor, and then display
DISPLAY cur_name TO info
```



```

-- this name to the information
-- field. The first cursor you
-- use in the program will be
-- given number 0. This number
-- will be increased by 1
-- every time you declare and
-- open a new cursor
-- during the program execution

LET i = 1 -- This is necessary for the first table row to be displayed to
-- the first line of the screen array below

WHILE TRUE -- starting a loop
    FETCH c_cur_1 INTO r_clients.* -- using the FETCH statement in a loop
    -- to retrieve multiple rows
    IF status = NOTFOUND
        THEN EXIT WHILE
    END IF
    -- when all the rows are retrieved from
    -- the table and no more rows can be
    -- found by the cursor, we exit the loop

# We retrieve the rows one by one and display them to the screen array
# in the order of retrieval
    DISPLAY r_clients.id_client, r_clients.first_name, r_clients.last_name,
          r_clients.start_date TO s_recl[i].*
    SLEEP 1
    LET i = i+1
# This will clear the screen array, which contains only five elements,
# and prepare it for displaying the rest of the rows fetched by the cursor

    IF i > 5 THEN
        FOR i = 1 TO 5
            CLEAR s_recl[i].*
        END FOR
        LET i = 1
    END IF
END WHILE

    CALL fgl_winmessage("Finished", "All rows have been retrieved", "info")
    CLOSE c_cur_1 -- We close the cursor
    CLOSE WINDOW cur_window

COMMAND "Selected Records"
    "Selecting the rows matching the WHERE clause from one table using a cursor"
# Cursor c_cur_2 is declared for a SELECT statement
# retrieving the records of the clients registered from "09/12/2010" to
# 09/18/2010"
OPEN WINDOW cur_window AT 2,2 WITH form "cur_form"
LET curs_name = "c_cur_2" -- we assign the cursor name to a variable
DECLARE curs_name CURSOR FOR -- and then use this variable in
                             -- the DELCARE statement
    SELECT c.*
    FROM clients c
    WHERE c.start_date BETWEEN DATE("09/12/2010") AND DATE("09/18/2010")

OPEN curs_name
LET cur_name = cursor_name("curs_name") -- We call the cursor_name

```



```

        -- function again
        -- in order to find out the name
        -- of the cursor assigned by the
        -- program

DISPLAY cur_name TO info

LET i = 1
WHILE TRUE
    FETCH curs_name INTO r_clients.*

        IF status = NOTFOUND THEN
            EXIT WHILE
        END IF

        DISPLAY r_clients.id_client, r_clients.first_name,
               r_clients.last_name,r_clients.start_date TO s_rec1[i].*
        SLEEP 1
        LET i = i + 1

        IF i > 5 THEN
            FOR i = 1 TO 5
                CLEAR s_rec1[i].*
            END FOR
            LET i = 1
        END IF
    END WHILE

CLOSE curs_name
CALL fgl_winmessage("Finished", "All rows have been retrieved", "info")
CLOSE WINDOW cur_window

COMMAND "Simple Join"
    "Retrieving rows from several tables with a simple join and a cursor"
# Cursor c_cur_3 is declared for a SELECT statement retrieving rows from
# three tables joined by a simple joint.
# The data is retrieved only for those clients which have the deposit
# contracts. The receiving records will obtain no NULL values
OPEN WINDOW cur_window AT 2,2 WITH FORM "cur_form"
DECLARE c_cur_3 CURSOR FOR
    SELECT t1.* ,t2.* ,t3.*
    FROM clients t1,contracts t2,country_list t3 -- simple join without OUTER

    WHERE t1.id_client = t2.id_client AND -- table clients is joined with
                                                 -- table contracts using id_client
                                                 -- columns
          t1.id_country = t3.id_country -- table clients is joined with
                                             -- table country_list using
                                             -- id_country columns

OPEN c_cur_3
LET cur_name = cursor_name ("c_cur_3")
DISPLAY cur_name TO info
LET i = 1
WHILE TRUE -- the INTO clause contains three records - each for one table
    FETCH c_cur_3 INTO r_clients.* ,r_contracts.* ,r_country_list.*

        IF status = NOTFOUND THEN
            EXIT WHILE

```



```

        END IF
        # Then we sort the data and display them for each loop iteration
        DISPLAY r_clients.id_client,r_clients.first_name,r_clients.last_name,
               r_clients.start_date,r_country_list.country_name TO s_rec2[i].*
              

        DISPLAY r_contracts.num_con,r_contracts.account,r_contracts.amount,
               r_contracts.interest_rate,r_contracts.start_date,
               r_contracts.end_date TO s_rec3[i].*
        SLEEP 1
        LET i = i + 1
        IF i > 5 THEN
            FOR i = 1 TO 5
                CLEAR s_rec2[i].*
                CLEAR s_rec3[i].*
            END FOR
            LET i = 1
        END IF

    END WHILE
    CLOSE c_cur_3
    CALL fgl_winmessage("Finished", "All rows have been retrieved", "info")
    CLOSE WINDOW cur_window

COMMAND "Outer Join"
"Retrieving rows from several tables with an outer join"
OPEN WINDOW cur_window AT 2,2 WITH FORM "cur_form"

# Cursor c_cur_4 is declared for a SELECT statement, which FROM clause
# contains the OUTER keyword.
# The SELECT statement selects the data from three interconnected tables:
# clients, contracts and country_list.
# The data are retrieved for all the clients regardless of whether they
# have a deposit contract.
DECLARE c_cur_4 CURSOR FOR
    SELECT t1.*,t2.*,t3.*
    FROM clients t1,country_list t2,OUTER contracts t3
    # clients table has simple join with country_list tabl
    # but it has an outer joint with contracts table
    WHERE t1.id_country = t2.id_country AND -- clients is joined with
                                                 -- country_list using
                                                 -- id_country fields
          t1.id_client   = t3.id_client      -- clients is joined with
                                                 -- contracts using an outer
                                                 -- join
    LET cur_name = cursor_name ("c_cur_4")
    DISPLAY cur_name TO info

    OPEN c_cur_4
    LET i = 1
    WHILE TRUE
        # In some cases r_contracts record will be empty, when the cursor
        # returns the client records for which there are no corresponding
        # deposit contracts records.
        FETCH c_cur_4 INTO r_clients.* ,r_country_list.* ,r_contracts.* 

```



```
IF status = NOTFOUND THEN
    EXIT WHILE
END IF

DISPLAY r_clients.id_client,r_clients.last_name,
       r_country_list.country_name,r_contracts.num_con,
       r_contracts.account,r_contracts.amount,
       r_contracts.interest_rate TO s_rec4[i].*
SLEEP 1
LET i = i + 1

IF i > 5 THEN
    FOR i = 1 TO 5
        CLEAR s_rec4[i].*
    END FOR
    LET i = 1
END IF

END WHILE
CLOSE c_cur_4
CALL fgl_winmessage("Finished", "All rows have been retrieved", "info")
CLOSE WINDOW cur_window

COMMAND "CURSOR Variable"
"Retrieving rows by means of a variable of the CURSOR data type"
OPEN WINDOW cur_window AT 2,2 WITH form "cur_form"
# The last cursor is associated with a variable of the CURSOR data type we
# defined at the beginning of the program.
# First, we need to declare a cursor by calling the Declare() method and
# passing a SELECT statement to it as the argument.
CALL c_cur_5.Declare("SELECT t1.*, t2.*, t3.* FROM clients t1, contracts
                      t2, country_list t3 WHERE t1.id_client = t2.id_client
                      AND t1.id_country = t3.id_country")

CALL c_cur_5.Open() -- Then we open the cursor

DISPLAY c_cur_5.GetName() TO info -- To find the program-assigned name of
-- the cursor, we call the GetName()
-- method this time

# Next, we specify the variables to hold the data fetched by the cursor
# as the arguments of the SetResults() method
CALL c_cur_5.SetResults(r_clients.* , r_contracts.* , r_country_list.*)
LET i = 1

WHILE (c_cur_5.FetchNext() = 0) -- The WHILE loop will continue until
-- the FetchNext() method retrieves all the
-- rows from the tables one by one, that is
-- until the value of sqlca.sqlcode changes
-- from 0 to 100(notfound), provided
-- no error occurs

# We display the rows to the screen record in the order of retrieval
DISPLAY r_clients.id_client, r_clients.last_name,
```



```

        r_country_list.country_name, r_contracts.num_con,
        r_contracts.account,r_contracts.amount,
        r_contracts.interest_rate TO s_rec4[i].*
SLEEP 1
LET i = i+1
IF i > 5 THEN

    FOR i = 1 TO 5
        CLEAR s_rec4[i].*
    END FOR
    LET i = 1
END IF
END WHILE

CALL c_cur_5.Close()-- we close the cursor
CALL c_cur_5.Free() -- and free it in order to release the
-- resources allocated to it.
-- Now you cannot reference the
-- cursor by any statements, and you
-- will have to declare and open it again

CALL fgl_winmessage( "Finished", "All rows have been retrieved", "info")
CLOSE WINDOW cur_window

COMMAND "Exit" "Exit the program"
EXIT PROGRAM

END MENU

END MAIN

```

The Form File

This is the form "cur_form.per" to which the values retrieved by the cursors are displayed.

```

DATABASE db_examples
SCREEN
{
    [client_lab]
\g-----
\g
    [f01      ][f02          ][f03          ][f04          ][f05          ]
    [f06]    [f07          ][f08          ][f09          ][f10          ]
\g-----
\g
    [contracts_lab]
\g-----
\g
    [f11      ][f12          ][f13          ][f14          ][f15          ][f16          ]
    [f17      ][f18          ][f19          ][f20          ][f21          ][f22          ]

```



```
[f17 ][f18 ][f19 ][f20 ][f21 ][f22 ]

\g-----
\g
      [f23 ][f24 ]
\g-----
\g
}
```

TABLES

clients contracts country_list

ATTRIBUTES

```
client_lab = formonly.client_lab, widget="label", config= "CLIENTS", COLOR =
BLUE BOLD UNDERLINE;
f01 = formonly.id_lab, widget="label", config= "Client ID", CENTER, COLOR = BLUE
BOLD ;
f02 = formonly.f_name_lab, widget="label", config= "First name", CENTER, COLOR =
BLUE BOLD;
f03 = formonly.l_name_lab, widget="label", config= "Last name", CENTER, COLOR =
BLUE BOLD;
f04 = formonly.clients_start_date_lab, widget="label", config= "Start date",
CENTER, COLOR = BLUE BOLD;
f05 = formonly.country_name_lab, widget = "label", config = "Country Name",
CENTER, COLOR = BLUE BOLD ;
f06 = clients.id_client, CENTER, COLOR = BOLD;
f07 = clients.first_name, CENTER,COLOR = BOLD;
f08 = clients.last_name, CENTER,COLOR = BOLD;
f09 = clients.start_date,CENTER, COLOR = BOLD;
f10 = country_list.country_name, CENTER, COLOR = BOLD;
contracts_lab = formonly.contracts_lab, widget="label", config= "CONTRACTS",
COLOR = BLUE BOLD UNDERLINE;
f11 = formonly.num_con_lab, widget = "label", config = "Contract Number",
CENTER, COLOR = BOLD BLUE;
f12 = formonly.account_lab, widget = "label", config = "Account", CENTER,
COLOR = BOLD BLUE;
f13 = formonly.amount_lab, widget = "label", config = "Amount", CENTER,
COLOR = BOLD BLUE;
f14 = formonly.interest_rate_lab, widget = "label", config = "Interest Rate",
CENTER, COLOR = BOLD BLUE;
f15 = formonly.contracts_start_date_lab, widget = "label",
config = "Start Date", CENTER, COLOR = BOLD BLUE;
f16 = formonly.end_date_lab, widget = "label", config = "End Date", CENTER,
COLOR = BOLD BLUE;
f17 = contracts.num_con,CENTER,COLOR = BOLD;
f18 = contracts.account, CENTER,COLOR = BOLD;
f19 = contracts.amount,CENTER,COLOR = BOLD;
f20 = contracts.interest_rate,CENTER, COLOR = BOLD;
f21 = contracts.start_date, CENTER, COLOR = BOLD;
f22 = contracts.end_date, CENTER, COLOR = BOLD;
f23 = formonly.info_lab, widget="label", config="CURSOR NAME:",
```



```
COLOR = BOLD BLUE UNDERLINE;
f24 = formonly.info, CENTER, COLOR = GREEN BOLD;

INSTRUCTIONS
DELIMITERS "[ ]"

SCREEN RECORD s_rec1 [5] (clients.id_client, clients.first_name,
                           clients.last_name, clients.start_date)
SCREEN RECORD s_rec2 [5] (clients.id_client, clients.first_name,
                           clients.last_name, clients.start_date,
                           country_list.country_name)
SCREEN RECORD s_rec3 [5] (contracts.num_con, contracts.account,
                           contracts.amount, contracts.interest_rate,
                           contracts.start_date, contracts.end_date)
SCREEN RECORD s_rec4 [5] (clients.id_client, clients.last_name,
                           country_list.country_name, contracts.num_con,
                           contracts.account, contracts.amount,
                           contracts.interest_rate)
```



Database Transactions

When an application is interacting with a database, it either retrieves the data from the database or modifies the database structure and/or data stored in it. These processes can be interrupted for unexpected reasons, such as hardware failure or incorrect data inputted by a user. In case of data retrieval the only consequence of that will be the error in your program execution which can be handled easily: the methods of error handling are described in the previous chapter. If a database modification process (UPDATE, INSERT statements, etc.) is terminated with an error, the error will occur in both your application and in the database. When a modification is interrupted by an external cause, you cannot be sure how much of the operation was completed. Even in a single-row operation, you cannot know whether the data reached the disk.

In order to ensure the integrity of the database data, the statements that affect the database structure and data stored in it can be used within database transactions. In such case, if an error occurs, all the changes performed during the current transaction can be undone.

The Notion of Transaction

A transaction is a sequence of database modifications that are regarded as a unity. Thus they can be either accomplished completely, or not at all. Thus a transaction is a set of statements limited by the special keywords, all of which need to be executed successfully for the transaction to be complete. The database server ensures that the operations within a transaction are completely committed to disc without errors, or the database is restored to the state in which it has been before the beginning of the transaction. A transaction also offers a program way to escape when a logical error occurs.

To be able to use the transactions and to restore the state of the database in case of an error, the database should allow transaction logging. Otherwise it will be impossible to rollback the changes, if the transaction fails, because the state of the database before the transaction will not be recorded.

Transaction Logging

The database server can keep a record of each change that it makes to the database during a transaction. If something happens to cancel the transaction, the database server automatically uses the records to reverse the changes.

The process of keeping records of transactions is called transaction logging. The records of the transactions are stored in a portion of disk space separate from the database. The transaction logging is available only if the database was initially created with the logging option. To allow the data logging, use the corresponding options when using tools for creating data bases.

You can also use the MODE ANSI log option to specify that the database is created in the ANSI mode.

Cancelling Transaction Logging for Temporary Tables

As you do not need to worry about the database integrity when working with temporary tables, you do not need the transactions to work with them. However, if a temporary table is created in a database which supports logging, any modifications in this table will also be recorder in the log file. This may cause some inconvenience, as the logging may require some time and disc space.



You can cancel the logging for the temporary table at the moment when it is created. Using the WITH NO LOG keywords in the CREATE TEMP TABLE statement prevents logging of temporary tables in databases started with logging.

If it is used, but the database does not use logging, the WITH NO LOG option is ignored. Once you turn off logging on a temporary table, you cannot turn it back on; a temporary table is, therefore, always logged or never logged.

The following example shows how to prevent logging for temporary tables in a database that uses logging:

```
CREATE TEMP TABLE tab2 (fname CHAR(15), lname CHAR(15))
WITH NO LOG
```

A temporary table is usually deleted, when the database in which the table is created is closed by the application and another database is opened. If no other database is opened, the temporary table remains in the database until the application is terminated. However, the table is deleted, if the database that does transaction logging is closed and the table declaration does not include the WITH NO LOG option.

Specifying the Transactions

There are explicitly specified transactions and implicit transactions; it depends on the database type.

In databases created with the MODE ANSI option transactions are started automatically as soon as the database is opened. However, you still need to mark the end of the transaction to be able to roll back the database state. Otherwise all manipulations will become the elements of one and the same transaction and in case if an error occurs, you will need to roll back all the modifications and not just the part which caused the error.

If the database is created without the MODE ANSI option, you need to mark the beginning and the end of a transaction explicitly, otherwise no transaction will be created.

The special statements are used to mark the beginning and the end of the transaction as well as for rolling back the changes in case of an error.

Starting a Transaction

To start a transaction explicitly you need to use the BEGIN WORK statement. This statement does not have any additional clauses and does not reference any variables, thus it consists only of the BEGIN WORK keywords.

```
BEGIN WORK
UPDATE items SET item_price = item_price * 1.10
WHERE manu_code = 'KAR'
DELETE FROM items WHERE manu_code = 'SEU'
...
```

You can issue the BEGIN WORK statement only if a transaction is not in progress. If you issue a BEGIN WORK statement while you are in a transaction, the database server returns an error. Thus you must not use this statement in a database created in MODE ANSI in which a transaction is always in progress.

A warning is generated, if you use a BEGIN WORK statement immediately after one of the following statements: DATABASE, COMMIT WORK, CREATE DATABASE, ROLLBACK WORK, and START WORK. An error is generated if you use a BEGIN WORK statement after any other statement.



Completing a Transaction

The COMMIT WORK statement is used to complete the transaction and to commit all the modifications made since the beginning of the transaction to the database. As well as the BEGIN WORK statement, this statement consists only of the COMMIT WORK keywords.

```
BEGIN WORK
UPDATE items SET item_price = item_price * 1.10
WHERE manu_code = 'KAR'
DELETE FROM items WHERE manu_code = 'SEU'
...
COMMIT WORK
```

You should use the COMMIT work at the end of a multi-statement operation when you are sure that you want to keep the changes made since the beginning of the transaction. In a database created with MODE ANSI the COMMIT WORK statement is necessary to mark the end of the transaction. It also marks the beginning of the next transaction, as a transaction starts automatically as soon as the previous one ends. In a not ANSI compliant database you can use the COMMIT WORK statement only after the BEGIN WORK statement. Thus to start another transaction after closing the previous one, the BEGIN WORK statement should follow the COMMIT WORK statement.

The COMMIT WORK statement has the following effects:

- Releases all locks imposed on the database tables (the locks are described later in this chapter).
- Closes all cursors open at the moment of its execution except for the cursors with hold which are discussed later in this chapter.
- Commits all the database changes performed since the beginning of the current transactions
- Completes the transaction (and begins a new implicit transaction, if it is an ANSI compliant database)

Rolling Back Changes

The ROLLBACK WORK statement can appear within a transaction, if you do not want to commit the changes that occurred since the beginning of the transaction.

In a database that is not ANSI-compliant, a transaction is started with a BEGIN WORK statement. You can end a transaction with a COMMIT WORK statement or cancel the transaction with a ROLLBACK WORK statement. The ROLLBACK WORK statement restores the database to the state that existed before the transaction began. In an ANSI compliant database a new transaction begins right after the ROLLBACK WORK statement, just like with the COMMIT WORK statement.

You cannot use both the COMMIT WORK and ROLLBACK WORK statements to complete one and the same transaction. You need to use them conditionally. Use the COMMIT WORK statement if no errors occurred, otherwise use the ROLLBACK statement.

Use the ROLLBACK WORK statement only at the end of a multi-statement operation after you have specified all the SQL statements you want to include into the current transaction. If you issue a ROLLBACK WORK statement when no transaction is pending for a not ANSI compliant database, an error occurs; in an ANSI compliant database the statement is accepted, but has no effect.

The ROLLBACK WORK statement has the following effects:



- Releases all locks imposed on the database tables (the locks are described later in this chapter).
- Closes all cursors open at the moment of its execution except for the cursors with hold which are discussed later in this chapter.
- Undoes all the database changes except those that are executed and committed to the database automatically immediately after execution (i.e. GRANT, CREATE TABLE, ALTER TABLE, CREATE DATABASE, DATABASE, etc.)
- Terminates the transaction (and begins a new implicit transaction, if it is an ANSI compliant database)

If you use the ROLLBACK WORK statement within a routine that a WHENEVER statement calls, specify WHENEVER SQLERROR CONTINUE and WHENEVER SQLWARNING CONTINUE before the ROLLBACK WORK statement. This prevents the program from looping if the ROLLBACK WORK statement encounters an error or a warning.

Locks

If your database is contained in a single-user workstation, without a network connecting it to other computers, concurrency is unimportant. In all other cases, you must allow for the possibility that, while your program is modifying data, another program is also reading or modifying the same data. Unless controls exist on the use of data, however, concurrency can lead to a variety of negative effects. Programs could read obsolete data; modifications could be lost even though it seems they were entered successfully.

To prevent this from happening you can use locks to secure the part of the database you plan to modify and to prevent the other database users to modify the same data at the same time. A lock is a reservation on a piece of data which allows only the program which locked it to modify it. If another program tries to address the locked data, an error is issued.

Locking the Whole Database

You can add the EXCLUSIVE keyword to the DATABASE statement, if you want the database to be accessed only by you. If this keyword is included, the database cannot be accessed by anyone, but the current user. This keyword can appear only in the specification of the [current database](#) - that is in the MAIN or FUNCTION program block. You cannot specify the [default database](#) as EXCLUSIVE. To cancel the exclusive mode you need to close the database using the CLOSE DATABASE statement and then reopen the database with another DATABASE statement. Here is an example of a current database opened in the exclusive mode:

```
MAIN
DEFINE...
DATABASE my_database EXCLUSIVE
```

If the database is already open in the exclusive mode by another user, you will not be able to connect to it and will receive an error, if you try. Because locking a database reduces concurrency in that database to zero, it makes programming very simple; concurrent effects cannot happen. However, you should lock a database only when no other programs need access to it.

Locking Tables

You can lock tables within a database to avoid locking the entire database. Sometimes a table is locked automatically. For example, it is done when the LATER TABLE statement is processed. The lock on a table is released either when the statement finished executing or when a transaction ends.



You can lock a table explicitly using the LOCK TABLE statement. The LOCK TABLE statement can lock a table either in share mode, which allows read only access for all other users, but denies the write access for all the users except the one who executed this statement, or in exclusive mode, which denies both write and read access for all the other users. Here is the syntax of this statement:

```
LOCK TABLE table_name IN SHARE | EXCLUSIVE MODE
```

You can lock a table if you own the table or have the select privilege on the table or on a column in the table. The LOCK TABLE statement fails if the table is already locked in exclusive mode by another process, or if an exclusive lock is attempted while another user has locked the table in share mode.

```
BEGIN WORK
LOCK TABLE items IN SHARE MODE
UPDATE items SET item_price = item_price * 1.10
WHERE manu_code = 'KAR'
DELETE FROM items WHERE manu_code = 'SEU'
...
COMMIT WORK
```



Note: If your database supports transactions, the LOCK TABLE statement can only be used within a transaction.

You cannot switch between share and exclusive mode within the transaction. Once you lock the table using this statement, the lock and its mode remain unchanged until the transaction ends.

Releasing the Lock

In a database without transactions the lock is released when:

- The UNLOCK TABLE statement is executed. It has the following syntax:

```
UNLOCK TABLE table_name
```

- The database is closed with the CLOSE DATABASE statement or with another DATABASE statement
- The application is terminated.

However, in a database with transactions, the lock is released, when the program executes the COMMIT WORK or ROLLBACK WORK statement, thus you do not need to release the lock explicitly. Moreover, the UNLOCK table will cause an error, if executed for a database with transactions.

Locking Rows

When the database server opens an update cursor, it places a promotable lock on all the fetched rows. If this action succeeds, the database server knows that no other program can alter those rows. Because a promotable lock is not exclusive, other programs can continue to read the rows. This helps performance because the program that fetched the row can take some time before it issues the UPDATE or DELETE statement, or it can simply fetch the next row.



When a row is about to be modified, an exclusive lock is put on it. If a row has a promotable lock on it, its status is changed to exclusive. If the database does not support transactions, the lock is released as soon as the row is modified. If the database uses transactions, the lock is released when the transaction is compete.

A Cursor with Hold

Any cursor type can be declared with hold. To declare a cursor with hold you need to add the WITH HOLD keywords to the DECLARE statement immediately after the CURSOR keyword:

```
DECLARE cursor_name [ SCROLL ] CURSOR WITH HOLD...
```

For more information about the types of the cursors and for the complete syntax of the DECLARE statement see chapters "[The Notion of Cursor](#)" and "[Advanced Cursor Usage](#)".

A hold cursor remains open after a transaction ends unlike the normal cursor which is closed either by the COMMIT WORK or ROLLBACK WORK statement. A cursor declared with hold allows you to access the same set of rows using multiple transactions.

If your database has transactions and the cursor was declared by DECLARE FOR UPDATE but not WITH HOLD, the FOREACH statement must be executed within a transaction. You can open an update cursor that was declared with a DECLARE WITH HOLD via a FOREACH statement outside a transaction, but you cannot roll back any changes that the cursor performs outside the transaction. In this situation, each UPDATE WHERE CURRENT OF is automatically committed as a singleton transaction.

A hold cursor can be closed using the CLOSE statement, the CLOSE DATABASE statement, it is not closed at the end of a transaction automatically.

Declaring a Cursor WITH HOLD Using a CURSOR Variable

To declare a cursor with hold while using a CURSOR variable, you should add the optional argument to the Declare() method. It should be located after the SQL statement text and after one more optional argument which will be discussed in the following chapter:

```
cursor_variable.Declare("SQL statement" [,with_scroll][,with_hold] )
```

The third argument is accepting a BOOLEAN value, specifies whether it will be a cursor with hold. By default this value is FALSE, so if you omit this argument, the cursor will be declared without hold. The following line declares a cursor with hold which selects all the rows from table *tab_1*.

```
CALL cur.Declare ("SELECT*FROM tab_1", FALSE, TRUE)
```

If you need to specify the with hold attribute, you need to specify the scroll attribute also. If only one Boolean value is specified, it will be treated as the scroll attribute. For now set the with scroll attribute to FALSE.

Declaring an Update Cursor

Use the FOR UPDATE keywords to declare an update cursor. You can use the update cursor to modify (update or delete) the current row. In such a way you can update or delete multiple rows.

```
DECLARE cursor_name CURSOR FOR SELECT_statement  
FOR UPDATE [OF column_list]
```



After you create an update cursor, you can update or delete the currently selected row by using an UPDATE or DELETE statement with the WHERE CURRENT OF clause. The words CURRENT OF refer to the row that was most recently fetched; they take the place of the usual test expressions in the WHERE clause. The UPDATE and DELETE statements with that clause are discussed later in this chapter.

The SELECT statement associated with the cursor sticks to the same rules as described for the read-only cursor above. When you declare a cursor with the FOR UPDATE keywords, all the rows conditioned by the associated SELECT statement are locked. This means that no other user than you can modify the locked rows.

Like with the read-only cursor, the DECLARE statement declaring a cursor for update is not an active statement. It just gives the name to the cursor, allocates memory for it, and assigns a SELECT statement to the cursor. The update cursor needs to be used in other statements in order to produce some effect.

You should use the cursor declared for update within a database transaction. Using it outside a transaction may cause database errors.

The OF Clause

When you use this kind of cursor for update, you can limit the columns to be modified by including the OF clause followed by the list of columns to be updated. The columns should be separated by commas. This clause is valid only if the cursor will be used to update the rows. The OF clause has no effect, if the cursor is used in a DELETE statement.

If the OF clause is present, only the columns specified in it become locked and impossible for other users to change. On the other hand, you can change only those columns which are specified by the OF clause.

You can modify only those named columns in subsequent UPDATE statements. The columns of the OF clause need not be in the select list of the SELECT clause. Here is an example of a cursor declared for update:

```
DECLARE curs1 CURSOR FOR
    SELECT * INTO my_rec.* FROM tab_1
    FOR UPDATE OF col_1, col_2
```

Updating Multiple Rows

To update multiple rows with one UPDATE statement, you should use a cursor declared for update. Updating database data is done using the following algorithm:

- Declare the cursor for update as described above
- Open the cursor with the OPEN statement
- Open a conditional loop where you will place the FETCH and UPDATE statement
- Use the FETCH statement within the conditional loop to retrieve the rows
- Then use the UPDATE statement to update the rows fetched within the same conditional loop. You can place the UPDATE statement in a conditional statement (IF or CASE), if you want to update only some of the fetched rows and perform some other actions on the other fetched rows.

The syntax of the UPDATE statement which uses the cursor is slightly different from the [UPDATE](#) statement which updates only one row and which has already been discussed earlier. It is as follows:



```
UPDATE table_name SET column [,column] = value [,value]
    WHERE CURRENT OF cursor_name
```

As you can see, the WHERE clause which would normally contain the condition for the update is replaced with the WHERE CURRENT OF keywords which are obligatory, if you want the UPDATE statement to use the cursor. In such a way the UPDATE statement will update any row where the cursor is currently located. The UPDATE statement does not advance the cursor to the next row, so the current row position remains unchanged. To advance the cursor to the new position, the FETCH statement needs to be executed.

The columns which can be updated by such UPDATE statement may be restricted if the DECLARE statement declaring the cursor specified includes the OF clause. The UPDATE statement can update only the columns included into the OF clause of the DECLARE statement. If the OF clause is omitted, any table column can be updated.

NOTFOUND Condition

If the FETCH statement seeks beyond the end of the current active set (which means beyond the set of the rows selected by the SELECT statement associated with the cursor), the *sqlca.sqlcode* variable is set to 100, which can be also represented by the NOTFOUND condition, when used in conditional statements. This typically happens when the FETCH statement fetches the last row of the active set and then attempts to fetch another row though no rows are left. This happens because the WHILE loop with *sqlca.sqlcode* = 0 condition is terminated only after the *sqlca.sqlcode* is set to 100, this means that the empty row is already retrieved.

If you try to process the empty row which returned the NOTFOUND condition (to update or delete it), you will receive an error. To be on the safe side, you can include the test which verifies whether the *sqlca.sqlcode* = 100 after the FETCH statement but before the UPDATE or DELETE statement to prevent them from processing this empty row:

```
WHILE TRUE
    FETCH cur
    IF sqlca.sqlcode = NOTFOUND THEN
        EXIT WHILE
    END IF
    ...
END WHILE
```

The example below illustrates how multiple rows can be updated:

```
# First we declare a cursor for update and restrict the columns
# which can be updated using the OF clause
DECLARE my_cur CURSOR FOR
    SELECT * FROM items WHERE item_id > 100
    FOR UPDATE OF price
OPEN my_cur --then we open the cursor

# We open a WHILE loop which checks the validity of the OPEN
# statement
WHILE sqlca.sqlcode = 0
    FETCH my_cur INTO upd_rec.* --here we fetch the rows one by one
```



```
# We place the UPDATE statement after the FETCH statement to
# process the rows fetched
    UPDATE items SET price = price*1,05
        WHERE CURRENT OF my_cur
    DISPLAY upd_rec.item_id, upd_rec.item_name, upd_rec.price
        AT 2,2
    SLEEP 2
END WHILE
CLOSE my_cur -- we close the cursor which we will need any more
FREE my_cur -- and free it
```

Deleting Multiple Rows

To delete multiple rows with one DELETE statement, you should use a cursor declared for update. The algorithm for deleting is pretty much the same as for updating with the only difference that the DELETE statement is used instead of the UPDATE. You also need to include the DELETE statement following the FETCH statement into a conditional loop, as the DELETE statement by itself does not move the cursor. After the deletion, no current row exists; you cannot use the cursor to delete or update a row until you reposition the cursor with a FETCH statement.

Here is the syntax of the DELETE statement used with the cursor which is a bit different from the general [DELETE](#) statement which deletes only one row:

```
DELETE FROM table_name WHERE CURRENT OF cursor_name
```

The WHERE CURRENT OF keywords replace the WHERE clause, but unlike the original WHERE clause, they are obligatory, if you want to use a cursor for deleting rows. The DELETE statement with the WHERE CURRENT OF clause will delete any row at which the cursor is currently located. If the DELETE statement is not placed within a conditional statement, it will delete all the fetched rows.

The example below is a somewhat modified variant of the example above which updates rows matching the condition and deletes those which do not:

```
DECLARE my_cur CURSOR FOR
    SELECT * FROM items
    FOR UPDATE
OPEN my_cur

WHILE sqlca.sqlcode = 0
    FETCH my_cur INTO upd_rec.*
    IF sqlca.sqlcode=NOTFOUND THEN
        EXIT WHILE
    END IF
    IF upd_rec.item_id > 100 THEN
        UPDATE items SET price = price*1,05
            WHERE CURRENT OF my_cur
    ELSE
        DELETE FROM items WHERE CURRENT OF my_cur
    END IF
END WHILE
```



Example

This application illustrates how the transactions work both when successful and when the rollback is needed. It also illustrates the additional types of a select cursor and the cursor declared with hold.

```
#####
# The transactions with the BEGIN WORK, COMMIT WORK and ROLLBACK WORK statements
# as well as the cursors WITH HOLD and FOR UPDATE
#####
DATABASE db_examples
DEFINE
    r_clients      RECORD LIKE clients.*,
    arr_clients    DYNAMIC ARRAY OF RECORD
        id_client  LIKE clients.id_client,
        first_name  LIKE clients.first_name,
        gender      LIKE clients.gender,
        start_date   LIKE clients.start_date,
        end_date     LIKE clients.end_date
    END RECORD,
    r_contracts   RECORD LIKE contracts.*,
    i_row,i,
    error_flag,
    num_key       INTEGER,
    cur_name      VARCHAR(30)

MAIN
# This option won't let the user terminate the application by pressing
# the close button
OPTIONS ON CLOSE APPLICATION KEY accept

# In order to operate normally, the program should be launched in GUI mode
# The following IF statement terminates the program, if it is launched in
# the character mode
IF fgl_fglgui() = FALSE
    THEN
        CALL fgl_winmessage("Character mode",
            "You tried to launch the program in character mode." ||
            "\nSome form widgets cannot be displayed and used correctly." ||
            "\nThe program will be terminated." ||
            "\n\nPlease, run the program in GUI mode", "error")
        EXIT PROGRAM
    END IF

MENU "main"

COMMAND "BEGIN WORK"
"Using the BEGIN WORK and COMMIT WORK for an explicit transaction"
CALL beg_com()

COMMAND "ROLLBACK"
"Using the ROLLBACK WORK statement to undo the transaction effect"
CALL rollback_work()

COMMAND "FOR UPDATE"
"Modifying the data in the table with a cursor FOR UPDATE"
CALL forupdate()
```



```
COMMAND "WITH HOLD"
"Declaring a cursor WITH HOLD and using it in a transaction"
CALL withhold( )

COMMAND "CURSOR VARIABLE"
"Using a cursor with hold declared as a variable of the CURSOR data type"
CALL cur_vars( )

COMMAND "EXIT"
"Exit the demo program"
EXIT PROGRAM

END MENU

END MAIN

#####
#The function beg_com() demonstrates how the BEGIN WORK and COMMIT WORK
#statements can be used to perform an explicit transaction
#####

FUNCTION beg_com( )

# We initialize the arr_clients to NULL because we need an empty array at the
# beginning of the function execution

    INITIALIZE arr_clients TO NULL

# The cursor is declared outside a transaction because
# the cursor declaration itself does not affect database data.
# This cursor is not used in the transaction below, but it is used further.
# In general, the DECLARE statements should better be placed at the beginning
# of the module, for they are similar to the DEFINE statements and are used
# to declare.

    DECLARE c_sel_clients CURSOR FOR
        SELECT c.* FROM clients c

# Here the transaction begins. This transaction doesn't use a cursor.
# It causes the UPDATE and INSERT operators to be either executed
# successfully or to be cancelled altogether, if an error occurs.
BEGIN WORK
    LET i = 10
    # We add 200 new records into the table and change the value of gender
    # column afterwards.
    WHILE TRUE
        LET i = i + 1
        IF i > 210 THEN EXIT WHILE END IF
        # We assign data to be inserted during each iteration
        LET r_clients.id_client    = i
        LET r_clients.id_country   = "840"
        LET r_clients.first_name   = "Lycia_",i USING "<<<&"
        LET r_clients.last_name    = "Lastt_name_",i USING "<<<&"
```



```

LET r_clients.date_birth = DATE("07/01/1980") + 1 UNITS DAY
LET r_clients.gender = "male"
LET r_clients.id_passport = "0000000",i USING "&&&"
LET r_clients.address = "New_address_",i USING "<<<&&@"
LET r_clients.start_date = TODAY
LET r_clients.end_date = DATE("12/31/9999")
# And insert them into the table
INSERT INTO clients VALUES(r_clients.*)
# Right after that we change the value just inserted for gender column
LET r_clients.gender = "female"
UPDATE clients SET clients.gender = r_clients.gender
    WHERE clients.id_passport = r_clients.id_passport
END WHILE
COMMIT WORK -- Here the transaction is completed. The validity of the changes
-- made from the beginning of the transaction is verified
-- and the data are recorded into the database

# After the transaction is completed, we use the cursor declared earlier
# to fill a program array with the results of the transaction
LET i_row = 1

OPEN c_sel_clients
WHILE TRUE

    FETCH c_sel_clients INTO r_clients
    IF status = NOTFOUND THEN EXIT WHILE END IF

    LET arr_clients[i_row] = r_clients.id_client, r_clients.first_name,
        r_clients.gender, r_clients.start_date,
        r_clients.end_date
    LET i_row = i_row+1

END WHILE
CLOSE c_sel_clients

CLEAR SCREEN

# Now we can display the values from the database to the screen array
OPEN WINDOW win1 AT 1,1 WITH FORM "update_display"
DISPLAY "!" TO formonly.back
DISPLAY "200 records were added to the table. Press BACK to continue . . ."
TO headr
CALL set_count(220)
DISPLAY ARRAY arr_clients TO scr_clients.*
ON ACTION (back)
    EXIT DISPLAY
END DISPLAY

# Then we delete the records we've just added to the clients table
DELETE FROM clients WHERE clients.id_client > 10

CLOSE WINDOW win1
CLEAR SCREEN

```



```
END FUNCTION

#####
#The function rollback_work() demonstrates how the ROLLBACK WORK statement
#can be used to undo the transaction effect
#####

FUNCTION rollback_work()
    INITIALIZE arr_clients TO NULL
    DECLARE c_sel_clients CURSOR FOR
        SELECT c.* FROM clients c

    LET error_flag = 0          -- we will use this variable to monitor the value of
                                -- the status variable
    WHENEVER ERROR CONTINUE   -- we instruct the program to execute further, even
                                -- if there is an error
    BEGIN WORK -- and begin a transaction which is supposed to cause an error
        LET i = 10
        WHILE TRUE
            LET i = i + 1
            IF i > 210 THEN EXIT WHILE END IF
            LET r_clients.id_client = i
            LET r_clients.id_country = "840"
            LET r_clients.first_name = "Lycia_",i USING "<<<<&"
            LET r_clients.last_name = "Lastt_name_",i USING "<<<<&"
            LET r_clients.date_birth = DATE("07/01/1980") + 1 UNITS DAY
            LET r_clients.gender = "male"
            LET r_clients.id_passport = "0000000",i USING "&&&"
            LET r_clients.address = "New_address_",i USING "<<<<&"
            LET r_clients.start_date = TODAY
            LET r_clients.end_date = DATE("12/31/9999")

            INSERT INTO clients
                VALUES(r_clients.*)
                -- if the INSERT fails, status variable
                -- acquires a negative value different from 0
            IF status <> 0
                -- we check whether the status variable has 0
                -- value after each iteration
            THEN LET error_flag = status
                # If it doesn't, the transaction is rolled back. All the changes
                # are cancelled.
                ROLLBACK WORK
                GOTO lab_1 -- the program control is passed beyond the
                            -- transaction
        END IF

        #If the IF statement above is not triggered, the transaction continues
        IF i = 60
        THEN # We assign an unacceptable value "sex" for the 50th record
            # which must cause an error
            LET r_clients.gender = "sex"
        ELSE LET r_clients.gender = "female"
        END IF

        # If the update fails, the status variable will also get a non-zero
```



```
# value
UPDATE clients SET clients.gender = r_clients.gender
    WHERE clients.id_passport = r_clients.id_passport

# We check the value of status again after the UPDATE.
IF status <> 0
THEN LET error_flag = status
    # If it is not 0, the transaction is rolled back. All the changes
    # are cancelled.
    ROLLBACK WORK
    GOTO lab_1 -- and the program control is passed beyond the
                -- transaction
END IF
END WHILE
COMMIT WORK           -- we complete the transaction
CLOSE c_sel_clients
WHENEVER ERROR STOP   -- and change the behaviour for errors

# This is the place where the program control is passed
# in case the transaction is rolled back
# as well as after it is completed successfully

LABEL lab_1:
# we check whether the transaction was a success
# and if it wasn't, we display a message about the error
IF error_flag <> 0
THEN call fgl_winmessage("Error!",

    "Owing to some data error the transaction rollback has been performed." ||
    "\nAll the modifications in the data base have been cancelled.",
    "Error")

END IF

# We search for the client data and display them to the screen
# You will see that no new records appeared, because the transaction was
# cancelled using the ROLLBACK WORK statement.
LET i_row = 1
OPEN c_sel_clients
WHILE TRUE

    FETCH c_sel_clients INTO r_clients
    IF status = NOTFOUND THEN EXIT WHILE END IF

    LET arr_clients[i_row] = r_clients.id_client, r_clients.first_name,
                            r_clients.gender, r_clients.start_date,
                            r_clients.end_date
    LET i_row = i_row+1

END WHILE
CLOSE c_sel_clients
```



```
# Then we delete the records we've just added
DELETE FROM clients WHERE clients.id_client > 10

# Now we can display the values taken from the database to the screen array
OPEN WINDOW win2 AT 1,1 WITH FORM "update_display"
OPTIONS FORM LINE 1
DISPLAY "!" TO formonly.back
DISPLAY "No values have been added to the table."
TO headr
CALL set_count(15)
DISPLAY ARRAY arr_clients TO scr_clients.*
ON ACTION (back)
    EXIT DISPLAY
END DISPLAY

CLOSE WINDOW win2
CLEAR SCREEN

# You can comment the WHENEVER ERROR CONTINUE and WHENEVER ERROR STOP
# statements.
# Then recompile and run the program. The result will be different,
# regardless of the error.

END FUNCTION

#####
# The withhold() function demonstrates how a cursor with hold can be
# declared and used within a transaction
#####
FUNCTION withhold()

# To make sure that the cursor does not close after the transaction is
# complete, we declare it WITH HOLD

DECLARE c_with_hold CURSOR WITH HOLD FOR
    SELECT c.* FROM clients c

# You can comment out the WITH HOLD keywords in the DECLARE statement above,
# compile the program and watch its reaction.
# The first row will be added, then the cursor will be closed by the COMMIT
# WORK statement and any subsequent updates will fail due to the absence of
# the cursor

LET i = 0
OPEN c_with_hold -- we open the cursor
WHILE TRUE -- and we begin the WHILE loop outside the transaction
    # If the WITH HOLD keywords are commented the attempt to use the cursor
    # after the transaction ends will fail
    # The code below handles this exception
    WHENEVER ERROR CONTINUE
        FETCH c_with_hold INTO r_clients.*
        LET error_flag = status
        IF error_flag = NOTFOUND
```



```
THEN EXIT WHILE
ELSE IF error_flag < 0
    THEN DISPLAY "SQL statement error number ",error_flag
        USING "-<<<&","." AT 22,1 ATTRIBUTE(RED, BOLD)
    DISPLAY "Fetch attempted on unopen cursor."
        AT 23,1 ATTRIBUTE(RED, BOLD)
    DISPLAY "Press any key to continue . . ."
        AT 24,1
    LET num_key = fgl_getkey()
    CLEAR SCREEN
    EXIT WHILE
END IF
END IF
WHENEVER ERROR STOP

# We specify data to be assigned to the new records
# for the contracts table
LET i = i + 1
LET r_contracts.id_contract      = 0
LET r_contracts.id_type          = 1
LET r_contracts.id_client        = r_clients.id_client
LET r_contracts.id_currency       = "840"
LET r_contracts.num_con           = i USING "&&&&"
LET r_contracts.account          = "263030900",r_clients.id_client
                                USING "&&&&"
LET r_contracts.amount            = 5000.00
LET r_contracts.interest_rate     = 12.05
LET r_contracts.start_date        = TODAY
LET r_contracts.end_date          = DATE("12/31/9999")

BEGIN WORK -- then we begin a transaction within the WHILE loop
# within the transaction we insert values into a table
INSERT INTO contracts VALUES(r_contracts.*)
# and update the values
UPDATE contracts
    SET contracts.amount = 7000.00, contracts.interest_rate = 22.00
    WHERE contracts.account = r_contracts.account
COMMIT WORK -- and close the transaction within the same loop
-- it will be started and then closed in each iteration

# After the transaction is closed, the WHILE loop continues to execute,
# and the FETCH statement works normally because cursor c_with_hold was
# declared WITH HOLD
END WHILE
CLOSE c_with_hold

DISPLAY "Declaring the cursor WITH HOLD and using it in a transaction"
AT 1,2 ATTRIBUTE(GREEN, BOLD)

DECLARE c_sel_contracts CURSOR FOR
SELECT c.* FROM contracts c

# We search and display the data of the deposit contracts
LET i_row = 2
OPEN c_sel_contracts
```



```

WHILE TRUE
    FETCH c_sel_contracts INTO r_contracts.*
    IF status = NOTFOUND THEN EXIT WHILE END IF
    DISPLAY r_contracts.id_client      USING "###", "  ",
        r_contracts.num_con,
        r_contracts.amount           USING "#####.&&.", "  ",
        r_contracts.interest_rate   USING "#&.&&.", "  ",
        r_contracts.start_date, "  ",
        r_contracts.end_date AT i_row,2
    LET i_row = i_row + 1
    IF i_row > 23
    THEN LET i_row = 2
        CLEAR SCREEN
        DISPLAY "Declaring the cursor WITH HOLD and using it in a transaction"
            AT 1,2 ATTRIBUTE(GREEN, BOLD)
    END IF
END WHILE
CLOSE c_sel_contracts
DISPLAY "Press any key to continue . . ." AT 24,1
LET num_key = fgl_getkey()
CLEAR SCREEN
# We delete the records we have just added
DELETE FROM contracts WHERE contracts.id_contract > 4

END FUNCTION

#####
# The forupdate() function is used to demonstrate how an update cursor can
# be declared and used within a transaction
#####

FUNCTION forupdate()

BEGIN WORK
# Cursor c_cur_5 is declared for a SELECT statement for update. All the rows
# retrieved by this cursor will be locked for the following modifications
# made using the UPDATE statement with the WHERE CURRENT OF option.
DECLARE c_cur_5 CURSOR FOR
    SELECT clients.* FROM clients FOR UPDATE

LET i_row = 2
OPEN c_cur_5
WHILE TRUE
    FETCH c_cur_5 INTO r_clients.*
    IF status = NOTFOUND THEN EXIT WHILE END IF
    LET i = i_row+1
    # We display r_clients record before the modifications
    DISPLAY "Before update:",r_clients.id_client USING "###", "  ",
        r_clients.add_data CLIPPED, "  ",
        r_clients.end_date AT i_row,2

    # Then we change the values of the columns add_data and end_date of
    # clients table.
    # The values are changed in the row in which the cursor is currently
    # located. After that the cursor moves to the next row.

```



```
UPDATE clients
    SET add_data = "Update of field add_data",          -- we update the column for
                                -- additional data
        end_date = TODAY                                -- and the column for the
                                -- end date
    WHERE CURRENT OF c_cur_5                            -- at the cursor location

    # Then we display r_clients record after the modification
    SELECT c.* INTO r_clients.* FROM clients c
        WHERE c.id_client = r_clients.id_client
    DISPLAY "After update:",r_clients.id_client USING "###", " ",
            r_clients.add_data CLIPPED," ",r_clients.end_date AT i,2
    LET i_row = i_row + 2
    IF i_row > 23 THEN LET i_row = 2 END IF
END WHILE
CLOSE c_cur_5
DISPLAY "Press any key to continue . . ." AT 24,1
LET num_key = fgl_getkey()
CLEAR SCREEN

# Here we restore the initial values of the fields updated above
UPDATE clients SET add_data = "N/A",end_date = "12/31/9999"

DISPLAY "Modification of the data using the cursor with FOR UPDATE OF column"
AT 1,2 ATTRIBUTE(GREEN, BOLD)

# Cursor c_cur_6 is declared for a SELECT statement with FOR UPDATE OF
# clients.add_data option.
# All the rows retrieved by this cursor will be locked to enable the usage of
# the UPDATE statement with the WHERE CURRENT OF option.
# Thus you can modify only the value of add_data column, as the OF clause
# restricts the possibilities for update.

DECLARE c_cur_6 CURSOR FOR
    SELECT clients.* FROM clients FOR UPDATE OF clients.add_data

LET i_row = 2
OPEN c_cur_6
WHILE TRUE
    FETCH c_cur_6 INTO r_clients.*
    IF status = NOTFOUND THEN EXIT WHILE END IF
    LET i = i_row+1
    # We display record r_clients before the update
    DISPLAY "Before update:",r_clients.id_client USING "###", " ",
            r_clients.add_data CLIPPED AT i_row,2

    # Then we change the values of the columns add_data and end_date
    UPDATE clients SET add_data = "Update of field add_data"
        WHERE CURRENT OF c_cur_6

    # We display r_clients after the update
    SELECT c.* INTO r_clients.* FROM clients c
        WHERE c.id_client = r_clients.id_client
    DISPLAY "After update:",r_clients.id_client USING "###", " ",
```



```

        r_clients.add_data CLIPPED AT i,2
    LET i_row = i_row + 2
    IF i_row > 23 THEN LET i_row = 2 END IF
END WHILE
CLOSE c_cur_6
DISPLAY "Press any key to continue . . ." AT 24,1
LET num_key = fgl_getkey()
CLEAR SCREEN

UPDATE clients SET add_data = "N/A"

DISPLAY "Deleting rows from a table using the FOR UPDATE cursor"
    AT 1,2 ATTRIBUTE(GREEN, BOLD)

# We add two records to table clients which contain the data of two clients
# so that we could delete them afterwards not violating the database data
INSERT INTO clients
    VALUES(0,804,"Julian","Menson", "01/01/1980", "male", "12345",
          "10/01/1998", "10/01/2018", "Broadway. 100, app.56",
          "San Francisco",NULL,"N/A","09/14/2010","12/31/9999")
INSERT INTO clients (id_client, id_country, first_name, last_name,
                     id_passport, address, start_date, end_date)
    VALUES(0,804,"Jim", "Holiday", "87654321", "Bourbon 12, app.4",
           "09/14/2010", "12/31/9999")

# Cursor c_cur_7 is declared for a SELECT statement with the FOR UPDATE
# option. All the rows retrieved by this cursor will be locked.
DECLARE c_cur_7 CURSOR FOR
    SELECT clients.* FROM clients
    WHERE clients.id_client > 10 -- this cursor will retrieve only the newly
                                  -- created records
    FOR UPDATE

OPEN c_cur_7
WHILE TRUE
    FETCH c_cur_7 INTO r_clients.*
    IF status = NOTFOUND THEN EXIT WHILE END IF
    # We delete all the records found by the cursor one by one
    DELETE FROM clients WHERE CURRENT OF c_cur_7
END WHILE
CLOSE c_cur_7
DISPLAY "Press any key to continue . . ." AT 24,1
LET num_key = fgl_getkey()
CLEAR SCREEN

COMMIT WORK

END FUNCTION

#####
# The cur_vars() function is used to demonstrate how variables of the CURSOR
# data type can be used instead of the usual cursors.
#####
FUNCTION cur_vars()

```



```
DEFINE cv_sel_clients CURSOR -- we will use this cursor for adding rows

CALL cv_sel_clients.Declare ("SELECT c.* FROM clients c", FALSE, TRUE)
INITIALIZE arr_clients TO NULL

# First, we need to add some values to the clients table and then
# we will use a variable of CURSOR data type to retrieve these values
# from the data base

BEGIN WORK
  LET i = 10
  # We add 90 new records into the table and change the value of gender
  # column afterwards.
  WHILE TRUE
    LET i = i + 1
    IF i > 100 THEN EXIT WHILE END IF
    # We assign data to be inserted during each iteration
    LET r_clients.id_client      = i
    LET r_clients.id_country     = "840"
    LET r_clients.first_name     = "Lycia_",i USING "<<<<&"
    LET r_clients.last_name      = "Lastt_name_",i USING "<<<<&"
    LET r_clients.date_birth     = DATE("07/01/1980") + 1 UNITS DAY
    LET r_clients.gender         = "male"
    LET r_clients.id_passport   = "0000000",i USING "&&&"
    LET r_clients.address        = "New_address_",i USING "<<<<&"
    LET r_clients.start_date    = TODAY
    LET r_clients.end_date      = DATE("12/31/9999")
    # And insert them into the table
    INSERT INTO clients VALUES(r_clients.*)

  END WHILE
  COMMIT WORK

  LET i_row = 1

  # Here we open the variable of the CURSOR data type declared before
  # and use it to fetch the information from the database
  CALL cv_sel_clients.open()
  CALL cv_sel_clients.SetResults(r_clients)
  WHILE (cv_sel_clients.FetchNext()=0)

    LET arr_clients[i_row] = r_clients.id_client, r_clients.first_name,
                           r_clients.gender, r_clients.start_date,
                           r_clients.end_date
    LET i_row = i_row+1

  END WHILE

  CALL cv_sel_clients.close()-- The CURSOR variable is closed

  CLEAR SCREEN

  # Now we can display the values from the database to the screen array
  # The ein3 window contains a screen grid to where the values will be
```



```
# displayed
OPEN WINDOW win3 AT 1,1 WITH FORM "update_display"
DISPLAY "!" TO formonly.back
DISPLAY "Next" TO formonly.back
DISPLAY "90 records were added to the table. Press NEXT to continue ..."
TO headr
CALL set_count(100)
DISPLAY ARRAY arr_clients TO scr_clients.*
ON ACTION (back)
    EXIT DISPLAY
END DISPLAY
CLOSE WINDOW win3
CLEAR SCREEN

# Now we can delete the rows inserted earlier by this function
# and return the variables used for data retrieval; to their initial values
    DELETE FROM clients WHERE clients.id_client > 10

LET i_row=1
INITIALIZE arr_clients TO NULL

# Now we check whether all the records where id_client>10 have been deleted
# Here, we will use a cursor with hold declared as a CURSOR data type
# variable

CALL cv_sel_clients.open()
CALL cv_sel_clients.SetResults(r_clients)
WHILE (cv_sel_clients.FetchNext()=0)

    LET arr_clients[i_row] = r_clients.id_client, r_clients.first_name,
        r_clients.gender, r_clients.start_date,
        r_clients.end_date
    LET i_row = i_row+1

END WHILE
CALL cv_sel_clients.close()

CLEAR SCREEN

OPEN WINDOW win3 AT 1,1 WITH FORM "update_display"
DISPLAY "!" TO formonly.back
DISPLAY "Back" TO formonly.back
DISPLAY "90 records were deleted from the table. Press BACK to continue ..."
TO headr
CALL set_count(100)
DISPLAY ARRAY arr_clients TO scr_clients.*
ON ACTION (back)
    EXIT DISPLAY
END DISPLAY

CLOSE WINDOW win3
CLEAR SCREEN

END FUNCTION
```



The Form File

Here is the form file 'update_display.per':

DATABASE db_examples

The Script File

It is also advisable that you create a script file containing the following line:

transactions.?.toolbarvisible: false



This script option will hide the default toolbar that appears in the GUI mode and won't let the user terminate the program by means of the 'cancel' button. Otherwise some of the database data may be violated due to the unfinished transaction, if the user closes the program prematurely.

If you use this script, your program should be called "transactions".



Advanced Cursor Usage

This chapter will dwell upon additional possibilities offered by the cursor, such as inserting multiple rows into a table with the help of the insert cursor. It will also touch upon another way of fetching rows from a table which is considered to be more compact and convenient compared to the usage of the OPEN, FETCH, and CLOSE statements, together with some enhancements which can be applied to the SELECT statement when using it with a cursor and which can sort the retrieved data.

The FOREACH Statement

The FOREACH statement is an alternative for using the OPEN, FETCH, CLOSE and WHILE statements for processing the cursor. It applies a series of actions to each row of data that is returned from a query by a cursor and creates a loop in itself so that you do not need to use the WHILE statement to loop the process of row fetching explicitly.

Here is the syntax of the FOREACH loop:

```
FOREACH cursor_name [ INTO clause ] statement_list END FOREACH
```

The cursor used in the FOREACH statement must be declared by the DECLARE statement but it need not be opened. The FOREACH statement has the following effects:

- Opens the specified cursor
- Fetches the rows specified by the SELECT statement associated with the cursor one at a time
- Executes the statements in the statement list
- Closes the cursor after the last row has been fetched

Thus the FOREACH statement is an equivalent for using the OPEN, FETCH, and CLOSE statements. If the cursor used by this statement is not declared, a compile-time error occurs. The FOREACH statement can reference any types of cursors, but the rows will be processed only in sequential order. The FOREACH statement performs successive fetches until all the rows specified by the SELECT statement are retrieved.

The cursor is automatically closed when all the rows are fetched, or when the FETCH loop encounters the NOTFOUND condition (sqlca.sqlcode=100).

The INTO Clause

The INTO clause of the FOREACH statement is identical in syntax and purpose of the [INTO clause](#) of the FETCH statement and contains the list of variables which accept the values retrieved from a table. There must be only one INTO clause among the statements referencing the cursor, thus the INTO clause can be placed either in the SELECT statement associated with the cursor, or in the FOREACH statement referencing it. It cannot be present in both of them.

The Statement List

The statement list of the FOREACH loop usually contains statements which process the rows retrieved. This list can include the UPDATE and DELETE statements, if the cursor referenced was declared for update. It can also include the EXIT FOREACH keywords and CONTINUE FOREACH keywords.



If the cursor returns no rows, then no statements in this loop are executed, and program control is passed to the first statement that follows the END FOREACH keywords.

Here is an example of the FOREACH statement:

```
DECLARE c_orders CURSOR FOR
  SELECT * INTO list.* FROM orders, customers
  WHERE orders.customer_num = customers.customer_num
  FOREACH c_orders
    DISPLAY list.mem1, list.mem2, list.mem3
  END FOREACH
```

The Declaration Clause

The statement list can begin with a declaration clause which specifies the spot variables that will be used only within the current FOREACH loop. This declaration clause is similar to the other declaration clauses used within statements (e.g., WHILE statement). The example given below demonstrates how a declaration clause can be used

```
DECLARE cur CURSOR FOR
  SELECT * INTO rec.* FROM clients
  FOREACH cur
    DEFINE i INT
    FOR i = 1 TO 3
      DISPLAY rec.* AT i, 2
    END FOR
  END FOREACH
```

The CONTINUE FOREACH Keywords

When 4GL encounters the CONTINUE FOREACH keywords within the FOREACH loop, it interrupts processing of the current row, returns the program control to the beginning of the loop, and retrieves another row. Thus all the statements between the CONTINUE FOREACH and END FOREACH keywords are skipped. The CONTINUE FOREACH keywords need to be used in a conditional statement; otherwise the statements following it will never be executed.

The EXIT FOREACH Keywords

When 4GL encounters the EXIT FOREACH keywords, it interrupts processing of the current row, terminates the FOREACH loop and passes the program control to the first statement following the END FOREACH statement.



The END FOREACH Keywords

The END FOREACH keywords indicate the end of the FOREACH loop. When 4GL encounters them, it returns the program control to the beginning of the loop and executes it again, if there are still rows left to retrieve. If there are no rows left, it passes the program control to the first statement following the END FOREACH statement.

The Scroll Cursor

The cursor sub-type associated with the SELECT statement we looked at so far was a sequential cursor which retrieves the rows in a sequential order. The other sub-type of the select cursor is the scroll cursor. The scroll cursor can be used to fetch rows from a table in any sequence.

Whether the cursor is sequential or scroll should be specified during the cursor declaration. Unlike the sequential cursor sub-type, the scroll cursor can be associated only with the SELECT statement and never with the INSERT statement. The syntax of the DECLARE statement for a scroll cursor is as follows:

```
DECLARE cursor_name SCROLL CURSOR FOR SELECT_statement
```

As you can see the FOR READ ONLY / FOR UPDATE keywords are not allowed in this syntax. The scroll cursor can be used only for retrieving data from a database; it cannot be used for update.

To implement a scroll cursor, the database server creates a temporary table to hold the active set. With the active set retained as a table, you can fetch the first, last, or any intermediate rows as well as fetch rows repeatedly without having to close and reopen the cursor. The database server retains the active set for a scroll cursor in a temporary table until the cursor is closed.

The [SELECT](#) statement was described in the previous chapter. Here is an example of a scroll cursor declared:

```
DECLARE my_c SCROLL CURSOR FOR SELECT * FROM my_tab
```

Declaring a Cursor with SCROLL Using a CURSOR Variable

To declare a scroll cursor using a CURSOR variable, you should add one more parameter to the Declare() method. It is an optional parameter and it should be located after the SQL statement text, but before the "with_hold" argument, if it is present:

```
cursor_variable.Declare("SQL statement" [,with_scroll][,with_hold] )
```

The second argument – *with_scroll* – defines a cursor to be of the SCROLL type, i.e. the one which can select multiple rows in any order. This optional parameter is represented by a value of the BOOLEAN data type – TRUE (1) or FALSE (0). By default, a cursor is sequential, so if the second parameter is left empty or the FALSE (0) value is specified, records from a table will be retrieved in the order from the first to the last.

On the other hand, if the TRUE (1) value is assigned to the second parameter, rows from a table can be fetched in an arbitrary order with corresponding methods being called to perform this task. A SCROLL cursor is discussed later in the manual. The example below declares a scroll cursor, because if only one optional attribute is present, it is treated as the with scroll attribute.

```
CALL cur.Declare("SELECT*FROM tab_1", TRUE)
```

A cursor with scroll and with hold will look as follows:



```
CALL cur.Declare("SELECT*FROM tab_1", TRUE, TRUE)
```

Processing a Scroll Cursor

A scroll cursor can be processed properly only using the FETCH statement. Whereas you can use a cursor declared as a scroll cursor in a FOREACH statement, the rows will be still retrieved in sequential order regardless of the cursor sub-type. However, if you use the FETCH statement, you can fetch any row in the active set, either by specifying an absolute row position or a relative offset.

The cursor name in the FETCH statement can be preceded by one of the keywords which specify which row should be fetched next. These keywords can be used only with a cursor declared as a SCROLL CURSOR.

```
FETCH keyword cursor_name [ INTO clause]
```

The keywords which are used to specify the position of the next row to be fetched are as follows:

Keyword	Result
NEXT	retrieves the row which is next to the current row. The current row is the row which has been previously retrieved by this FETCH statement. This is the default value.
PREVIOUS/PRIOR	retrieves the previous row in the active set.
FIRST	retrieves the first row of the active set.
LAST	retrieves the last row of the active set.
CURRENT	retrieves the current row of the active set (the same row which was returned by the previously executed FETCH statement).
RELATIVE [+/-] n	retrieves the row which position is relative to the current cursor position. By default it returns the row which is equal to current_row+n where n is an integer or a variable of the INTEGER data type. If you specify (-), it will return the row which position is earlier by n than the current row.
ABSOLUTE n	retrieves the row which position is specified by n. The minimum absolute position of a row is 1. The maximum absolute position is the total number of records in the active set.

Here are some examples which use the scroll cursor:

```
FETCH CURRENT my_curl
FETCH FIRST my_cur2
FETCH RELATIVE -5 my_cur3 -- fetches the row which is 5 rows above
                           -- the current one
FETCH ABSOLUTE 10 my_cur4 -- fetches the 10th row of the active set
```

Processing a SCROLL Cursor Associated with a CURSOR Variable

If a SCROLL cursor was declared with a variable of the CURSOR data type, it is processed by means of specific methods equivalent of the keywords in the FETCH statement described above.

The methods, operations they perform and arguments they require are presented in the table that follows.



Notice that only one method – *FetchNext()* can be used with both types of cursors – scroll and sequential. The rest are typical of the scroll cursor only.

Method	Parameters	Operation
FetchNext()	No parameters	Retrieves the row which is next to the current one. The current row is the row which has been previously retrieved by this <i>FetchNext()</i> method. This is the default value
FetchFirst()	No parameters	Retrieves the first row of the active set
FetchLast()	No parameters	Retrieves the last row of the active set
FetchPrevious()	No parameters	Retrieves the previous row in the active set
FetchRelative ([+/-] n)	n is an integer or a variable of the INTEGER data type. If you specify (-), it will return the row whose position is earlier by n than the current row, if (+) - the row whose position is greater by n than the current row	Retrieves the row whose position is relative to the current cursor position.
FetchAbsolute (n)	An integer which represents the absolute position of the row	Retrieves the row the position of which is specified by n. The minimum absolute position of a row is 0. The maximum absolute position is the total number of records in the active set. If you specify the absolute position as 0, the method will retrieve data from the same row which was used by the previously executed method from this table (this is equivalent to the <i>FETCH CURRENT</i> statement).

All the methods in the table return a value of the *sqlca.sqlcode* global record member.

Here is an example of the methods being called:

```

DEFINE cur CURSOR,
id INT, first_name CHAR (20)
CALL cur.Declare("SELECT * FROM tab_1 ",TRUE) -- This is a SCROLL cursor
-- because the second
-- argument is specified as
-- TRUE

CALL cur.Open()
CALL cur.setResults(id, first_name)
CALL cur.FetchFirst() -- The first row of the active set is fetched
DISPLAY id, " ", first_name
SLEEP 1
CALL cur.FetchLast() - The last row of the active set is retrieved
DISPLAY id, " ", first_name
SLEEP 1
# This will fetch the result set which is three rows below the one
# retrieved by the previous method, FetchLast() in this case
CALL cur.FetchRelative(-3) DISPLAY id, " ", first_name

```



```
SLEEP 1
# This statement retrieves the fourth row of the active set
# irrespective of the result returned by the previous method -
# FetchRelative()

CALL cur.FetchAbsolute(4) DISPLAY id, " ", first_name
SLEEP 1
CALL cur.FetchAbsolute(0) -- This will fetch the current row, that is the
-- row returned by the FetchAbsolute(4) method

DISPLAY id, " ", first_name
SLEEP 1
# This statement will retrieve the row before the one returned by the
# previous method - FetchCurrent()
CALL cur.FetchPrevious()
DISPLAY id, " ", first_name
SLEEP 1

...
```

Advanced Select Options

The SELECT statement clause which retrieves multiple rows is associated either with a cursor or with such statements as CREATE VIEW or UNLOAD which allow multiple rows selection. It can include some additional sub-clauses which structure the retrieved rows. You cannot use these clauses in a stand-alone SELECT statement which retrieves a single row and does not use a cursor, because it is not possible to sort a single row.

Aggregate Expressions

Aggregate expressions use aggregate functions to summarize the selected database data. They can be used instead of the column names in the select list of a SELECT clause, if you want to select not the values of a column but a minimum or maximum value, a sum of values, etc.

The aggregate functions cannot be used as stand-alone built-in functions. They can be used only within the SELECT clause or statement. Some of them are also used in 4GL reports but the reports are discussed later in this manual.

There are the following aggregate functions:

Keyword	Result
AVG (column_name)	returns the average value of all the values stored in the column specified as its argument. E.g. SELECT AVG(price) FROM items. NULL values are ignored, unless every row contains NULL, then this function returns NULL.
MAX (column_name)	returns the largest value contained in the column specified as the function argument. E.g. SELECT MAX(price) FROM items - returns the highest price
MIN (column_name)	returns the smallest value contained in the column specified as the function argument. E.g. SELECT MIN(price) FROM items - returns the lowest price
SUM (column_name)	returns the sum total of all the values stored in the column specified as its argument. E.g. SELECT SUM(price) FROM items - returns the total price for all the items



Note: The above listed aggregate functions can be applied only to the columns of numeric data types.

An aggregate function returns one value for a set of queried rows. These aggregate functions can be used both for the stand-alone SELECT statement and for the SELECT clause. An aggregate function cannot contain another aggregate function as its argument.

The COUNT Function

The COUNT() aggregate function can be applied to any column. It does not perform some calculations using the values of the column, but calculates the number of values.

COUNT(*)	returns the number of rows that satisfy the WHERE clause of a SELECT statement. E.g. SELECT COUNT(*) FROM items WHERE item_price = 100.00 - this SELECT statement returns an integer which specifies how many rows from the items table contain value 100.00 in the item_price column. If the WHERE clause is omitted, this function returns the total number of rows in the table.
COUNT (column_name)	returns the total number of non-null values in the specified column.

The COUNT(column_name) aggregate function can include the DISTINCT or UNIQUE keyword which change the result returned. These keywords are synonyms and have the same effect. You can place one of them before the column name to make the function return only the number of unique values in the column. E.g.:

```
SELECT COUNT(DISTINCT items) FROM item_tab
```

The GROUP BY Clause

The GROUP BY clause can be included into a SELECT clause of the CREATE VIEW or UNLOAD statement to sort the retrieved rows. It can also be included into the SELECT statement associated with a scroll cursor, because to group the retrieved values the Select statement needs to form an active set and to retrieve rows in non-sequential order. However, it cannot be used for a sequential cursor declared either read-only or for update.

Use the GROUP BY clause to produce a single row of results for each group. A group is a set of rows that have the same values for each column listed. If must follow the WHERE clause, if it is present. This clause has the following syntax:

```
GROUP BY column_name [, column_name...]
```

The column name is the name of a stand-alone column in the select list of the SELECT clause or the name of one of the columns joined by an arithmetic operator in the select list.

This sub-clause will make the SELECT clause sort the retrieved rows and arrange them in as many groups as many unique values there are for the specified column. For example, if the table called "personal_data" were queried using the following SELECT clause, the data retrieved would be divided into two groups, one of which will contain "m" in the sex column for all the rows of the group and the other will contain "f" in that column:

```
SELECT * FROM pers_data GROUP BY sex
```



Note: You cannot use a DECLARE statement with a FOR UPDATE clause to associate a cursor with a SELECT statement that has an ORDER BY clause.

The following example uses one column that is not in an aggregate expression in the GROUP BY clause. The tot_price column should not be in the GROUP BY list because it appears as the argument of an aggregate function. The COUNT and SUM keywords are applied to each group separately, not the whole query set.

```
SELECT order_id, COUNT(*), SUM(tot_price)
FROM items
GROUP BY order_id
```

Using Numbers

In place of column names, you can enter one or more integers that refer to the relative position of items in the SELECT clause. This is useful when you want to order the retrieved values by an expression and not by a column name. As you can use expressions in the select list instead of columns, this option might prove useful.

The example below groups the retrieved values according to the value returned by the "price*1.25" expression:

```
SELECT item_id, item_amm, price*1.25 FROM items
GROUP BY 3
```

The HAVING Clause

The HAVING clause is often used in combination with the GROUP BY clause. It applies one or more qualifying conditions to groups formed by the BY GROUP clause. It has the following syntax:

HAVING condition

Here condition is a relative expressions or a combination of relative expressions with Boolean operators AND and OR. The HAVING clause generally complements a GROUP BY clause. If you use a HAVING clause without a GROUP BY clause, the HAVING clause applies to all rows that satisfy the query. Without a GROUP BY clause, all rows in the table make up a single group.

The following example returns the *customer_id*, *call_time* (in full year-to-fraction format), and *cust_code*, and groups them by *call_code* for all calls that have been received from customers with *customer_id* less than 100:

```
SELECT customer_id, call_time, call_code
FROM cust_calls
GROUP BY call_code, 2, 1
HAVING customer_num < 100
```

The ORDER BY Clause

The ORDER BY clause is used to sort query results by the values that are contained in one or more columns. This clause, just like the GROUP BY clause, can appear in the SELECT clauses returning multiple rows or in the SELECT statement associated with a scroll cursor. It has the following syntax and has to follow the HAVING and GROUP BY clauses, if they are present:



```
ORDER BY column_name [, column_name] [ASC | DESC]
```

The ASC and DESC keywords are optional and they specify whether the rows should be arranged in ascending or descending order. If the explicit order specification is omitted, the order is ascending (ASC): from A to Z and from 1 to 9.

The following example arranges the rows retrieved from the table in alphabetical descending order based on the value in the *lname* column. This means that the first row will most probably contain the value beginning with "Z" and the last will contain the value beginning with "A":

```
SELECT * FROM pers_data ORDER BY lname DESC
```

Like with the GROUP BY clause you can use the integers to specify the column you want to use for ordering. The number should specify the position of the column which must be used for sorting in the select list.

Ordering by Column Substring

You can order the rows not according the whole column value but according to a substring. The column substring is similar to the substring of a character value. The column substrings are specified in the same way as the character substrings - you should put the coordinates of the first and the last character of the substring in square brackets following the column name, e.g.:

```
SELECT * FROM pers_data ORDER BY lname[1,1] ASC
```

In this way the retrieved values will be ordered according to the first character of the column value, the other characters will be ignored and the rows where *lname* values begin with the same letter will be grouped together in the order in which they are physically located in the table, unless you add the second column to the ORDER BY clause to make the ordering nested as explained below.

Nested Ordering

If you list more than one column in the ORDER BY clause, your query will be ordered by a nested sort. This means that the database sorts the values in several steps. First the rows are sorted according to the values of the first column which follows the ORDER BY keywords immediately. Then the rows are sorted again according to the values in the second column specified and so on. This means that if after the first level of sorting there are some rows for which the values are the same in the first column, they will be arranged among themselves in the order implied by the second column.

The Insert Cursor

A sequential cursor can be associated with the INSERT statement. An INSERT statement cannot be associated with a scroll cursor. The insert cursor cannot be used for update or for performing queries to a database. It is used to insert multiple rows into a database table.

The syntax for declaring the insert cursor is as follows:

```
DECLARE cursor_name CURSOR FOR INSERT_statement
```

The INSERT Statement Associated with the Cursor

The INSERT statement associated with a cursor has the syntax of a standard INSERT statement:



```
INSERT INTO table_name [(column_list)] VALUES (values_list)
```

Here is an example of an insert cursor declaration:

```
DECLARE inp_cur CURSOR FOR
    INSERT INTO my_tab VALUES (variable1, variable2, variable3)
```

The VALUES clause of a normal INSERT statement can contain variables, expressions, or literal values. If there are literal values in the values list, they cannot be changed by the statements that constitute the cursor processing and thus the values inserted into the rows will be the same for every row. However, if you use variables in the VALUES clause, the values of these variables can be changed while processing the cursor and thus each row can obtain different values.

When all the values in the VALUES clause are literals (or constants), the PUT statement has a special effect. Instead of putting a row in the buffer each time it is processed, the PUT statement merely increments a counter. When you then empty the buffer and insert the rows into the table, one row and a repetition count are sent to the database server, which inserts that number of rows.

The Insert Cursor Created by Means of a CURSOR Variable

The INSERT statement for a cursor associated with a CURSOR variable is specified in the *Declare()* method called on that variable. Basically, the syntax of this INSERT statement is similar to that described in the previous section.

```
CALL variable_name.Declare("INSERT INTO table_name[(column_list)]
    VALUES (values_list)")
```

variable_name stands for the identifier of a CURSOR variable defined before.

The rules of the *column_list* specification are also similar to those described above.

If you associate the INSERT statement with the cursor specified as a variable, and all the values that are to be inserted are known and given in the *values_list*, the syntax remains the same as described in the previous section. The only difference is that you have to take the character values in double quotes (" ") or in single quotes (' ') according to the quotation rules described at the beginning of this manual:

```
DEFINE cur CURSOR
CALL cur.Declare("INSERT INTO tab_1 VALUES (7, ""Mary"", 'Johnson')")
```

If the *value_list* of this INSERT statement contains one or more values that are to be substituted during the program execution, you should use question mark placeholders (?) instead of the names of the variables that will be used for the substitution. You should put one question mark for each value which is to be inserted:

```
DEFINE cur CURSOR
CALL cur.Declare("INSERT INTO tab_1 VALUES (?, ?, ?)")
```



Note: The values list in this case cannot contain variables. It can contain either placeholders, or literal values.

It is clear, that we have to do something to specify which values are to be added to the INSERT statement instead of the question mark placeholders. The *SetParameters()* method is used for this purpose.



The *SetParameters()* method takes as many arguments as many question marks the INSERT statement of the corresponding cursor has. For the cursor, given above, the *SetParameters()* method can look as follows:

```
CALL cur.SetParameters(8, "Ashley", lname)
```

Here, the first value is an integer, the second value is represented by a character string, and the third value is to be taken from the variable *lname* specified earlier in the program. The principle of the parameters specification is close to the principle of the arguments passing to a function: the values passed to the INSERT statement should correspond to the placeholders in number, and they should correspond to the data types, requirements, and order of the columns to which they are to be inserted.

The INSERT statement is a standard SQL statement which can be represented either by a quoted character string or by a variable of a character data type. The following parts of the source code have the same effect:

```
DEFINE cur CURSOR
CALL cur.Declare("INSERT INTO tab_1 VALUES (?, ?, ?)")

or

DEFINE ins CHAR(300)
LET ins = "INSERT INTO tab_1 VALUES (?, ?, ?)"
DEFINE cur CURSOR
CALL cur.Declare(ins CLIPPED)
```

Processing the Insert Cursor

The insert cursor is a completely different type of cursor, if compared to the select cursor. The insert cursor cannot be used with either FETCH or FOREACH statements, because it cannot be used to retrieve rows from a table. The insert cursor is used with the PUT statement. It must still be opened by the OPEN statement to be used and closed by the CLOSE statement when not used any more.

The algorithm for using an insert cursor is as follows:

1. Declare the insert cursor using the DECLARE statement.
2. Open the cursor with the OPEN statement.
3. Create a conditional loop similar to the loop created for processing the FETCH statement for the select cursor. This can be a WHILE loop or the FOR loop, if you know how many rows you want to insert.
4. Within the conditional loop specify the statements that assign the values to the variables of the VALUES clause of the INSERT statement, if there are any.
5. Within the same loop specify the PUT statement which will send the values to the insert buffer.
6. Insert the rows stored in the buffer to the database using either the CLOSE statement or the FLUSH statement.

Processing an Insert Cursor Associated With a Variable of the CURSOR Data Type

The algorithm for using a cursor with a CURSOR variable is this:

1. Declare the insert cursor using the *Declare()* method.
2. Open the cursor with the *Open()* method
3. If needed, specify the values that are to be passed to the values clause of the INSERT statement by calling the *SetParameters()* method.
4. If you want to insert more than one row, create a conditional loop. Within this loop call the *Put()* method which will send the values to the insert buffer.
5. Insert the rows stored in the buffer to the database using either the *Close()* or the *Flush()* method.



Adding Rows to the Insert Buffer

The PUT statement is used to store a row in the insert buffer for later insertion into the table. The insert buffer is created when the insert cursor is opened using the OPEN statement or method. The syntax of the PUT statement is as follows:

```
PUT cursor_name
```

The cursor name must belong to the cursor previously declared and associated with an INSERT statement.

If the buffer has no room for the new row when the statement executes, the buffered rows are written to the database in a block and the buffer is emptied. As a result, some PUT statement executions cause rows to be written to the database, and some do not. If you want to write the rows to the database before closing the cursor and then use the cursor and the buffer again, you can use the FLUSH statement for that purpose.

Here is an example which declares a cursor associated with the INSERT statement and then uses a conditional loop to assign values to the variables in the values list and to add the rows of these values to the insert buffer:

```
DECLARE inp_cur CURSOR FOR
    INSERT INTO my_tab VALUES (variable1, variable2, "test")
FOR i = 1 TO 5
    LET variable1 = i
    LET variable2 = i+1
    PUT inp_cur
END FOR
```

The example above does not yet insert the rows into a table, but it adds five rows to the insert buffer. The first row in the buffer will have the following values: 1, 2, "test". Each time the loop is iterated, the values are changed and then sent to the insert buffer. However, each row inserted will contain "test" as the third value and it will not be changed.

Adding Rows to the Insert Buffer with a CURSOR Variable

To add rows to a database table from the insert buffer with a cursor declared by means of a CURSOR variable, use the *Put()* method - the equivalent of the PUT statement. The method takes no arguments and returns a value of sqlca.sqlcode.

```
CALL variable_name.Put()
```

Variable_name refers to the variable of the CURSOR data type declared earlier.

The example from the previous section modified for the use of a CURSOR variable looks like this:

```
DEFINE cur CURSOR
CALL cur.Declare("INSERT INTO my_tab VALUES (?, ?, 'test') ")
CALL cur.SetParameters(variable1,variable2)

FOR i = 1 TO 5
    LET variable1 = i
    LET variable2 = i+1
    CALL cur.Put()
```



END FOR

Inserting the Rows into the Database

To insert the rows stored in the insert buffer into the database you can use either CLOSE statement or FLUSH statement.

Closing an Insert Cursor

When you use the CLOSE statement to close an insert cursor, all the rows which are stored in the insert buffer at that moment are inserted into the database.

```
CLOSE cur
```

The number of rows that were successfully inserted is returned in the third element of the **sqlerrd** built-in array member of the **sqlca** built-in record (sqlca.sqlerrd[3]).

Closing an Insert Cursor Associated with a CURSOR Variable

An insert cursor declared by means of a CURSOR variable is closed with the *Close()* method called on that variable. The method requires no arguments and returns a value of sqlca.sqlcode

```
CALL cursor_variable.Close()
```

Flushing an Insert Cursor

The FLUSH statement is used to insert the rows stored in the insert buffer into the database while the cursor is not yet closed. A PUT statement adds a new row to the buffer each time it is executed. The buffer content is written to the database under the following circumstances:

- When the buffer is full and another PUT statement is executed. The buffer content is inserted into the database, because there is no place for a new row.
- When the program executes the CLOSE statement which inserts all the rows currently stored in the buffer into the database.
- When the program executes the FLUSH statement. In this case the buffer does not need to be full and the cursor does not need to be closed and can be used further on.

	Note: If the program terminates without executing either CLOSE or FLUSH statement, the rows in the insert buffer are not inserted into the table and are lost.
--	---

The syntax of the FLUSH statement is easy:

```
FLUSH cursor_name
```

The cursor name must belong to an insert cursor, this statement cannot be used with other types of the cursors.



Here is an example which illustrates the work of the CLOSE and FLUSH statements:

```
DECLARE inp_cur CURSOR FOR
    INSERT INTO my_tab VALUES (variable1, variable2, "test")
FOR i = 1 TO 15
    LET variable1 = i
    LET vatiabile2 = i+1
    PUT inp_cur
    IF variable1 = 5 THEN
        FLUSH ins_cur
    END IF
END FOR
CLOSE inp_cur
```

In this case, the insert buffer will be flushed once and then the processing of the insert cursor will continue. The next 10 rows will be inserted by the CLOSE statement.

Flushing an Insert Cursor with a CURSOR Variable

To flush the contents of the insert buffer when a variable of the CURSOR data type was used to create the INSERT cursor, call the *Flush()* method. It is the equivalent of the standard FLUSH statement and follows the same rules.

No parameters are required by *Flush()* and it returns a value of the sqlca.sqlcode global record member.

The example from the previous section can be changed to use a CURSOR variable and call the *Close()* and *Flush()* methods.

```
DEFINE cur CURSOR
CALL cur.Declare("INSERT INTO my_tab VALUES (?, ?, 'test')")
CALL cur.Open()
CALL cur.SetParameters (variable1, variable2)
FOR i = 1 TO 15
    LET variable1 = i
    LET vatiabile2 = i+1
    CALL cur.Put()
    IF variable1 = 5 THEN
        CALL cur.Flush()
    END IF
END FOR
CALL cur.Close()
```

SQL INTERRUPT Option

The execution of SQL statements can be interrupted by the user as well as the 4GL statements. This ability can be enabled or disabled by the SQL INTERRUPT ON/OFF clauses of the OPTION statement which specify whether the Interrupt key (CONTROL-\) can be used to interrupt SQL statements. The default setting for the SQL INTERRUPT option is OFF, but it can be changed in the following way:

```
MAIN
DEFINE...
OPTIONS SQL INTERRUPT ON
```



```
...
END MAIN
```

If the SQL interruption is enabled and the user presses the Interrupt key while an SQL statement execution (i.e. while the SELECT statement is performing a query), the program waits until the database server completes the SQL statement and then executes the interruption.

If the DEFER INTERRUPT statement isn't executed before the user presses the Interrupt key, 4GL terminates the program execution. If the DEFER INTERRUPT statement was activated, 4GL assigns the value TRUE to the built-in variable *int_flag*, and the program execution isn't terminated.

It can be useful to enable the SQL INTERRUPT option only for a certain part of the program, so you can disable it by another OPTIONS statement:

```
OPTIONS SQL INTERRUPT ON
...
...
...
OPTIONS SQL INTERRUPT OFF
```

Example

This example illustrates the insert cursor and the scroll cursor as well as sorting and grouping of records retrieved from a database. It also contains example of the aggregate functions both used with cursors and without them.

```
#####
# This application illustrates the profound usage of cursors of different types,
# the FOREACH statement and grouping and sorting in the SELECT statement
#####
DATABASE db_examples
```

```
DEFINE
  r_clients      RECORD LIKE clients.*,
  r_contracts    RECORD LIKE contracts.*,
  full_name      CHAR(30),
  i,
  num_key,
  num_rows       INTEGER, -- holds the number of rows the user wants to be
                        -- inserted into the table
  fetch_row,
  sel_agr        CHAR(30), -- stores the number of a SELECT cursor with the
                        -- GROUP BY and HAVING options
  max_min_index, -- these are used to determine which
  avg_sum_index, -- aggregate functions should
  count_f_index, -- be executed
  max_min,        -- stores the minimum or maximum value returned by the
                  -- MAX() and MIN() functions
  avg_sum,        -- stores values returned by the AVG() and SUM() functions
  count_f        SMALLINT, -- stores values returned by COUNT(*) and
                        -- COUNT(col_name) functions
  cur            CURSOR,   -- variable of the CURSOR data type
  answer         CHAR(3),
```



P STRING

MAIN

```
# This application uses widgets non-displayable
# in character mode
IF fgl_fglgui()= 0 THEN
  CALL fgl_winmessage( "Wrong mode",
    "Run this application in the GUI mode.", "info")
  EXIT PROGRAM
END IF

OPTIONS
  ACCEPT KEY F1,
  PROMPT LINE LAST -- the prompt will be hidden

MENU ""
COMMAND "Insert Cursor"
  "Inserting multiple rows"
  CALL ins_func()

COMMAND "Scroll Cursor"
  "Using the SCROLL CURSOR"
  CALL scroll_func()

COMMAND "Aggregate Functions (1)"
  "Simple aggregate functions without a cursor"
  CALL aggr_func1()

COMMAND "Aggregate Functions (2)"
  "Aggregate functions in a SELECT with GROUP BY and HAVING"
  CALL aggr_func2()

COMMAND "Exit" "Exit the program"
  EXIT PROGRAM

END MENU

END MAIN

#####
# In this function the INSERT cursor is used to insert multiple rows into the
# clients table, which are then displayed to the screen array
#####
FUNCTION ins_func()

OPEN WINDOW win_cur AT 2,2 WITH FORM "clients_form"
  DISPLAY "INSERT CURSOR" TO lab1
  DISPLAY "Id_Client" TO lab2
  DISPLAY "First Name" TO lab3
  DISPLAY "Last Name" TO lab4
  DISPLAY "Start Date" TO lab5
```



```

DISPLAY "End Date" TO lab6
DISPLAY "!" TO insert_b
DISPLAY "!" TO quit_b

# We use prompt in a loop because we don't have input here,
# but still need to use the ON KEY clauses
WHILE TRUE
PROMPT "" FOR CHAR p

ON ACTION ("insert")
FOR i = 1 TO 10
CLEAR s_rec[i].*
END FOR
LABEL lab1:
# The user is prompted to enter the number of rows they want
# to be added to the table
LET num_rows = fgl_winprompt(2,2,
    "Enter the number of rows to be inserted:", "", 5, 2)
IF num_rows IS NULL THEN
    LET answer = fgl_winquestion("NO VALUE",
        "You have entered no value." ||
        "\n Do you want to quit?", "Yes", "Yes|No", "question", 1)
    IF answer MATCHES "Yes" THEN
        CLOSE WINDOW win_cur
        GO TO lab2
    ELSE
        GO TO lab1
    END IF
END IF

BEGIN WORK -- we begin the transaction
LET i = 10
LET num_rows = 10 + num_rows
# We call the Declare() method on the variable of the CURSOR type
# we declared earlier, and specify the SQL query. The question marks stand
# for the variables that will receive their values at runtime
CALL cur.Declare ("INSERT INTO clients VALUES(?,?,?,?,?,?,?,?,?,?,?,?,?,?)")
    CALL cur.Open() -- We open the cursor
    CALL cur.SetParameters(r_clients.*)
        -- and define the variables that
        -- will pass their values to the
        -- Declare() method
WHILE status = 0 -- we use the conditional loop to insert a number of rows
    LET i = i + 1
    IF i > num_rows THEN EXIT WHILE END IF -- The number of new records to be
                                                -- inserted into the table
                                                -- is supplied by the user. We add
                                                -- this number to the initial
                                                -- number of rows, which is 10.
                                                -- The loop will terminate when
                                                -- the number of rows is 10 plus
                                                -- the number provided by the user
                                                -- in the prompt field
    # We assign the data to be inserted into the table at each loop iteration
    LET r_clients.id_client = i

```



```

LET r_clients.id_country = "840"
LET r_clients.first_name = "First_name_", i USING "<<<<&"
LET r_clients.last_name = "Last_name_", i USING "<<<<&"
LET r_clients.date_birth = DATE("07/01/1980") + 1 UNITS DAY
LET r_clients.gender = "male"
LET r_clients.id_passport = "0000000", i USING "&&&"
LET r_clients.date_issue = r_clients.date_birth + 18 UNITS YEAR
LET r_clients.date_expire = NULL
LET r_clients.address = "New_address_", i USING "<<<<&"
LET r_clients.city = "New_city_", i USING "<<<<&"
LET r_clients.zip_code = "123", i USING "&&&"
LET r_clients.add_data = "N/A"
LET r_clients.start_date = TODAY
LET r_clients.end_date = DATE("12/31/9999")
CALL cur.Put() -- this statement places a row with the data to be
                -- inserted into the insert buffer nothing is
                -- inserted into the table yet
                -- at this point
END WHILE

IF status = 0 THEN
    CALL cur.Flush()
    IF status = 0
        THEN COMMIT WORK
        ELSE ROLLBACK WORK
        END IF
    ELSE ROLLBACK WORK
    END IF
    CALL cur.Close()
    -- We close the insert cursor. This does not insert
    -- any data into the table
    -- because the buffer was emptied by the FLUSH
    -- statement before.

DECLARE c_sel_clients CURSOR FOR
    SELECT c.* FROM clients c -- this is a select cursor
    WHERE c.id_client > 10 -- which retrieves only values inserted by the
                            -- insert cursor
    ORDER BY c.id_client

LET i = 0
FOREACH c_sel_clients INTO r_clients.* -- the FOREACH itself starts a loop
                                         -- and opens the cursor,
                                         -- so we need neither the WHILE
                                         -- statement like with OPEN and
                                         -- FETCH nor the OPEN statement

# Then we display the values retrieved by each loop iteration to each row of the
# screen array
DISPLAY r_clients.id_client, r_clients.first_name, r_clients.last_name,
       r_clients.start_date, r_clients.end_date TO s_rec[i].*
# When the screen array is full, and there are still some rows left in the

```



```

# program record, we clear the screen array and the remaining values are
# displayed to it.
        IF i = 10 THEN
            SLEEP 1
            FOR i = 1 TO 10
                CLEAR s_rec[i].*
            END FOR
            LET i = 0
        END IF
    END FOREACH -- the program will return to the beginning of the loop,
-- if there are still rows left to fetch
-- otherwise the cursor will be closed

# We delete the records we have inserted and then viewed
DELETE FROM clients WHERE clients.id_client > 10

ON KEY (F2)

LABEL lab2
EXIT WHILE

END PROMPT
END WHILE
CLOSE WINDOW win_cur
CLEAR SCREEN
END FUNCTION

#####
# This function uses the SCROLL cursor to retrieve multiple rows
# from the table in an arbitrary order
#####
FUNCTION scroll_func()
    OPEN WINDOW win_cur AT 2,2 WITH FORM "clients_form"
        DISPLAY "SCROLL CURSOR" TO lab1
        DISPLAY "Id_Client" TO lab2
        DISPLAY "First Name" TO lab3
        DISPLAY "Last Name" TO lab4
        DISPLAY "Start Date" TO lab5
        DISPLAY "End Date" TO lab6
        DISPLAY "FETCHED ROW:" TO lab10
        DISPLAY "Fetch a row:" TO lab11
        DISPLAY "!" TO f_row_but
        DISPLAY "!" TO quit_b

    INITIALIZE r_clients.* TO NULL
    # We declare a SELECT cursor and display the retrieved values to the screen
    # array, so that the user could see which row was fetched last
    DECLARE c_sel_clients CURSOR FOR
        SELECT c.* FROM clients c

    LET i = 0
    FOREACH c_sel_clients INTO r_clients.*
        LET i = i + 1
        DISPLAY r_clients.id_client, r_clients.first_name,
r_clients.last_name, r_clients.start_date,

```



```

        r_clients.end_date TO s_rec[i].*
    END FOREACH

# Here, we use the CURSOR type variable we declared at the beginning of the
# module. The Declare() method specifies the SQL query to be performed. The
# cursor is declared to be of the SCROLL type since the keyword TRUE is included
# as the second arguments
    CALL cur.Declare ("SELECT c.* FROM clients c ORDER BY c.id_client", TRUE)
    CALL cur.Open()-- Then the cursor is opened
    CALL cur.SetResults(r_clients.*)-- The results of the selection will be sent
                                    -- to the program record
    INPUT BY NAME fetch_row -- Then we activate the combobox field for input.
    ON KEY (F2)           -- The user can choose a row to be fetched by
                          -- selecting an appropriate list option
    EXIT INPUT

AFTER INPUT      -- When the Accept button is pressed, the case statement
                  -- determines which row will be fetched and displayed
CASE fetch_row -- at the bottom of the form
    WHEN "FETCH FIRST"
        DISPLAY "FIRST" TO row_id
        CALL cur.FetchFirst()-- This method fetches the first row of the
                            -- active set - the row in which id_client=1
        DISPLAY r_clients.id_client,r_clients.first_name,r_clients.last_name,
               r_clients.start_date,r_clients.end_date TO s_rec_fetch.*
        SLEEP 2
        CLEAR s_rec_fetch.* -- This clears the screen record to which the
                            -- fetched row was displayed
        CLEAR row_id         -- as well as the dynamic label
                            -- Initially, there are only three items in the
                            -- Combo box list.
        CALL populate_list()-- This function populates the list with the rest
                            -- of the options
    # fetches the last row of the active set - the row in which id_client=10
    WHEN "FETCH LAST"
        DISPLAY "LAST" TO row_id
        CALL cur.FetchLast()
        DISPLAY r_clients.id_client,r_clients.first_name,r_clients.last_name,
               r_clients.start_date, r_clients.end_date TO s_rec_fetch.*
        SLEEP 2
        CLEAR s_rec_fetch.*
        CLEAR row_id
        CALL populate_list()
    # Fetches the fifth row (where id_client=5)
    WHEN "FETCH ABSOLUTE 5"
        DISPLAY "ABSOLUTE 5" TO row_id
        CALL cur.FetchAbsolute(5)
        DISPLAY r_clients.id_client,r_clients.first_name,r_clients.last_name,
               r_clients.start_date,r_clients.end_date TO s_rec_fetch.*
        SLEEP 2
        CLEAR s_rec_fetch.*
        CLEAR row_id
        CALL populate_list()
    # Fetches the row which is 2 rows lower than the current one,
    # that is the one that was fetched last
    WHEN "FETCH RELATIVE +2"

```



```

        DISPLAY "RELATIVE +2" TO row_id
        CALL cur.FetchRelative(+2)
        DISPLAY r_clients.id_client,r_clients.first_name,r_clients.last_name,
               r_clients.start_date,r_clients.end_date TO s_rec_fetch.* 
        SLEEP 2
        CLEAR s_rec_fetch.*
        CLEAR row_id
# Fetches the row which is 4 rows above the current row
WHEN "FETCH RELATIVE -4"
        DISPLAY "RELATIVE -4" TO row_id
        CALL cur.FetchRelative(-4)
        DISPLAY r_clients.id_client,r_clients.first_name,r_clients.last_name,
               r_clients.start_date,r_clients.end_date TO s_rec_fetch.* 
        SLEEP 2
        CLEAR s_rec_fetch.*
        CLEAR row_id
# Fetching the row following the current one
# this is the default option also occurring in a non-scroll cursor
WHEN "FETCH NEXT"
        DISPLAY "NEXT" TO row_id
        CALL cur.FetchNext()
        DISPLAY r_clients.id_client,r_clients.first_name,r_clients.last_name,
               r_clients.start_date,r_clients.end_date TO s_rec_fetch.* 
        SLEEP 2
        CLEAR s_rec_fetch.*
        CLEAR row_id
# Fetching the previous row
WHEN "FETCH PREVIOUS"
        DISPLAY "PREVIOUS" TO row_id
        CALL cur.FetchPrevious()
        DISPLAY r_clients.id_client,r_clients.first_name,r_clients.last_name,
               r_clients.start_date,r_clients.end_date TO s_rec_fetch.* 
        SLEEP 2
        CLEAR s_rec_fetch.*
        CLEAR row_id
# Fetching the current row
WHEN "FETCH CURRENT"
        DISPLAY "CURRENT" TO row_id
        FETCH CURRENT c_scroll_clients INTO r_clients.*
        DISPLAY r_clients.id_client,r_clients.first_name,r_clients.last_name,
               r_clients.start_date,r_clients.end_date TO s_rec_fetch.* 
        SLEEP 2
        CLEAR s_rec_fetch.*
        CLEAR row_id

END CASE
CONTINUE INPUT
END INPUT
CALL cur.Close()-- We close the cursor
CALL cur.Free() -- And free it because we don't need it any more
CLOSE WINDOW win_cur
END FUNCTION

#####
# This function illustrates the use of aggregate functions in the SELECT

```



```

# statement without a cursor
#####
FUNCTION aggr_func1()
    OPEN WINDOW win_cur AT 2,2 WITH FORM "clients_form"
        DISPLAY "AGGREGATE FUNCTIONS 1" TO lab1
        DISPLAY "Interest Rate" TO lab3
        DISPLAY "Amount" TO lab4
        DISPLAY "Id_currency" TO lab5
        DISPLAY "Interest Rate:" TO lab7
        DISPLAY "Amount:" TO lab8
        DISPLAY "Count:" TO lab9
        DISPLAY "MIN" TO max_min_but
        DISPLAY "AVG" TO avg_sum_but
        DISPLAY "COUNT(*)" TO count_but
        DISPLAY "!" TO max_min_but
        DISPLAY "!" TO avg_sum_but
        DISPLAY "!" TO count_but
        DISPLAY "!" TO quit_b

# We declare a select cursor
DECLARE con_sel_cur
CURSOR FOR
SELECT con.* FROM contracts con
# and employ it to display the values that are going to be
# used in the aggregate functions, so that the user could
# follow what is happening
LET i = 0
FOREACH con_sel_cur INTO r_contracts.*
    LET i = i + 1
    DISPLAY r_contracts.interest_rate,
        r_contracts.amount,r_contracts.id_currency
        TO s_rec[i].field2, s_rec[i].field3,s_rec[i].field4
    IF i = 10 THEN
        LET i = 0
    END IF
END FOREACH
# We initialize the variables that serve to determine which
# aggregate function should be executed when a button is pressed
LET max_min_index = 0
LET avg_sum_index = 0
LET count_f_index = 0
# The fields to which the results of the aggregate functions execution
# will be displayed are activated for input
INPUT BY NAME max_min,avg_sum,count_f
    ON KEY (F2)
    EXIT INPUT
# Pressing the first button triggers either the MAX() or MIN() function,
# which return the maximum and minimum value in the interest_rate column
# correspondingly
ON ACTION ("Max_min")
    CASE max_min_index
        WHEN 0
            DISPLAY "MIN" TO max_min_but -- Initially, the MIN() function is
            -- executed

```



```

        SELECT MIN(interest_rate) INTO r_contracts.interest_rate
        FROM contracts
        DISPLAY r_contracts.interest_rate TO max_min
        SLEEP 2
        CLEAR max_min
        DISPLAY "MAX" TO max_min_but -- After the first button press,
        LET max_min_index = 1           -- the value is displayed to the
                                         -- field,
                                         -- the label on the button is changed,
                                         -- and the flag is set to 1
WHEN 1   -- The second key press will call the MAX() function
        SELECT MAX(interest_rate) INTO r_contracts.interest_rate
        FROM contracts
        DISPLAY r_contracts.interest_rate TO max_min
        SLEEP 2
        CLEAR max_min
        DISPLAY "MIN" TO max_min_but
        LET max_min_index = 0 -- When the maximum value is displayed to the
                           -- field the flag is set to 0 again

    END CASE

ON ACTION ("Avg_sum") -- When this action is activated, the
CASE avg_sum_index -- AVG() and SUM() functions are invoked
WHEN 0           -- to return the average value and the total sum
        DISPLAY "AVG" TO avg_sum_but -- of the values in the amount column
        SELECT AVG(amount) INTO r_contracts.amount FROM contracts
        DISPLAY r_contracts.amount TO avg_sum
        SLEEP 2
        CLEAR avg_sum
        DISPLAY "SUM" TO avg_sum_but
        LET avg_sum_index = 1
WHEN 1
        SELECT SUM(amount) INTO r_contracts.amount FROM contracts
        DISPLAY r_contracts.amount TO avg_sum
        SLEEP 2
        CLEAR avg_sum
        DISPLAY "AVG" TO avg_sum_but
        LET avg_sum_index = 0

    END CASE
ON ACTION ("Count")
CASE count_f_index
WHEN 0
        DISPLAY "COUNT(*)" TO count_but -- This will return
        SELECT COUNT(*) INTO i FROM contracts -- the total number of
        DISPLAY i TO count_f -- records in the table
        SLEEP 2
        DISPLAY "COUNT(id_currency)" TO count_but
        CLEAR count_f
        LET count_f_index = 1
WHEN 1 -- And this will display the number of unique currencies in
       -- the active set
        SELECT COUNT(DISTINCT id_currency) INTO i FROM contracts
        DISPLAY i TO count_f

```



```

        SLEEP 2
        DISPLAY "COUNT(*)" TO count_but
        CLEAR count_f
        LET count_f_index = 0
    END CASE
END INPUT
CLOSE WINDOW win_cur

END FUNCTION

#####
# This function illustrates the use of the SELECT cursor with the GROUP BY
# AND HAVING options
#####
FUNCTION aggr_func2()
    OPEN WINDOW win_cur AT 2,2 WITH FORM "clients_form"
    DISPLAY "AGGREGATE FUNCTIONS 2" TO lab1
    DISPLAY "Id_Client" TO lab2
    DISPLAY "Full Name" TO lab3
    DISPLAY "Id_currency" TO lab4
    DISPLAY "Interest Rate" TO lab5
    DISPLAY "Choose a function:" TO lab12
    DISPLAY "!" TO aggr_but
    DISPLAY "!" TO quit_b
    # The user can choose which aggregate function to be executed by
    # selecting a corresponding combo box list option
    INPUT BY NAME sel_agr
    ON KEY (F2)
    EXIT INPUT

    AFTER INPUT
    CASE sel_agr
        WHEN "Case 1"
            FOR i = 1 TO 10
                CLEAR s_rec[i].*
            END FOR
            CALL fgl_winmessage ( "INFORMATION",
                "Average interest rate for each currency," ||
                "\nwhich means grouping by currency id." ||
                "\nValues are sorted by the currency id" ||
                "\ndescending",
                "info" )

            # To retrieve the average interest rates for each currency
            # sorted in the ascending order by the currency codes
            # we need to declare the select cursor
            DECLARE c_set_contracts_1 CURSOR FOR
                SELECT id_currency, AVG(interest_rate) -- we add the aggregate
                                                -- function to the select
                                                -- list
            FROM contracts -- this option is required, if you have the
                            -- aggregate function
            GROUP BY id_currency -- in the select list, otherwise you will get
                                -- a runtime error
                                -- the aggregate function will calculate
    END CASE
END FUNCTION

```



```

-- values for each group
ORDER BY id_currency DESC -- the values will be ordered in the
-- descending order
LET i = 0
FOREACH c_set_contracts_1 INTO
    r_contracts.id_currency,r_contracts.interest_rate
    LET i = i + 1
    DISPLAY r_contracts.id_currency, r_contracts.interest_rate TO
        s_rec[i].field3,s_rec[i].field4
END FOREACH

WHEN "Case 2"
FOR i = 1 TO 10
    CLEAR s_rec[i].*
END FOR
CALL fgl_winmessage( "INFORMATION",
    "Average interest rate for each currency" ||
    "\nprovided that this average rate is > 10.00%",
    "info" )
# This cursor retrieves the average interest rate for each currency,
# only if the interest rate is greater than 10.00%
DECLARE c_set_contracts_2 CURSOR FOR
SELECT id_currency, AVG(interest_rate) -- this cursor retrieves the
-- average interest rate for
-- each currency
FROM contracts
GROUP BY 1 -- grouping the values by the currency codes
HAVING AVG(interest_rate) > 10.00
ORDER BY 1 DESC
LET i = 0
FOREACH c_set_contracts_2 INTO
    r_contracts.id_currency,r_contracts.interest_rate
    LET i = i + 1
    DISPLAY r_contracts.id_currency, r_contracts.interest_rate TO
        s_rec[i].field3,s_rec[i].field4
END FOREACH

WHEN "Case 3"
FOR i = 1 TO 10
    CLEAR s_rec[i].*
END FOR
CALL fgl_winmessage ( "INFORMATION",
    "Grouped by client id first," ||
    "\nthen by first and last name," ||
    "\nthen by currency id, displaying" ||
    "\nmax. interest rate for each client.",
    "info" )

DECLARE c_set_contracts_3 CURSOR FOR
# the concatenated columns are treated as one item
SELECT cli.id_client, cli.first_name||cli.last_name,
    con.id_currency, MAX(con.interest_rate) -- the maximum
-- interest rate for
-- each client
INTO r_clients.id_client,full_name,

```



```

        r_contracts.id_currency,r_contracts.interest_rate
    FROM clients cli,contracts con -- this time we select data from two
                                    -- tables
    WHERE cli.id_client = con.id_client -- using this join condition
    GROUP BY cli.id_client,-- grouping is done by client id first, so
            -- the max. interest rates
            -- are calculated for each client
            2,
            -- only a number can be used to group by an
            -- expression (cli.first_name||cli.last_name)
            cli.last_name,
            con.id_currency
    ORDER BY cli.id_client,con.id_currency
    LET i = 0
    FOREACH c_set_contracts_3
        LET i = i+1
        DISPLAY r_clients.id_client, full_name,
               r_contracts.id_currency,r_contracts.interest_rate TO
               s_rec[i].field1,s_rec[i].field2,s_rec[i].field3,
               s_rec[i].field4
    END FOREACH

    WHEN "Case 4"
        FOR i = 1 TO 10
            CLEAR s_rec[i].*
        END FOR
        CALL fgl_winmessage ( "INFORMATION",
                               "Max. interest rates for each client" ||
                               "\nwhere it is greater than 12.00% ",
                               "info")
        DECLARE c_set_contracts_4 CURSOR FOR
        SELECT cli.id_client,cli.first_name||cli.last_name,con.id_currency,
               MAX(con.interest_rate)-- the maximum interest rates for each client
               INTO r_clients.id_client,full_name,r_contracts.id_currency,
                     r_contracts.interest_rate
        FROM clients cli,contracts con
        WHERE cli.id_client = con.id_client
        GROUP BY 1,2,3,con.id_currency
        HAVING MAX(con.interest_rate) > 12.00 -- retrieves only interest
                                         -- rates > 12.00%
        ORDER BY 1,3
        LET i = 0
        FOREACH c_set_contracts_4
            LET i = i + 1
            DISPLAY r_clients.id_client,full_name,r_contracts.id_currency,
                   r_contracts.interest_rate TO
                   s_rec[i].field1,s_rec[i].field2,s_rec[i].field3,
                   s_rec[i].field4
        END FOREACH

    END CASE

    CONTINUE INPUT
    END INPUT
    CLOSE WINDOW win_cur

```



END FUNCTION

```
#####
# This function populates the combo box list with items
#####
FUNCTION populate_list()

    CALL fgl_list_set("fetch_row", 4, "FETCH RELATIVE +2")
    CALL fgl_list_set("fetch_row", 5, "FETCH RELATIVE -4")
    CALL fgl_list_set("fetch_row", 6, "FETCH NEXT")
    CALL fgl_list_set("fetch_row", 7, "FETCH PREVIOUS")
    CALL fgl_list_set("fetch_row", 8, "FETCH CURRENT")

END FUNCTION
```

The Form File

This is the form "clients_form.per"
DATABASE db_examples

SCREEN

```
{
    [ lab1 ] ]
\g-----
[ lab11      ][lab7      ][max_min_but ][max_min][lab12      ]
[ fetch_row   ][lab8      ][avg_sum_but  ][avg_sum][sel_agr     ]
    [f_row_but] [lab9      ][count_but   ][count_f  ][aggr_but   ]
\g-----
[ lab2      ][lab3      ][lab4      ][lab5      ][lab6      ]
[ field1    ][field2    ][field3    ][field4    ][field5    ]
\g-----
[ lab10      ] [row_id      ]
[ fld1      ][fld2      ][fld3      ][fld4      ][fld5      ]
    [insert_b  ]           [quit_b    ]
}
```

TABLES

clients contracts

ATTRIBUTES

```
lab1 = formonly.lab1, widget = "label", config = "", LEFT, COLOR = BLUE BOLD;
lab2 = formonly.lab2,widget = "label", config = "", CENTER, COLOR = BLUE BOLD;
lab3 = formonly.lab3, widget = "label", config = "", CENTER, COLOR = BLUE BOLD;
```



```
lab4 = formonly.lab4, widget = "label", config = "", CENTER, COLOR = BLUE BOLD;
lab5 = formonly.lab5, widget = "label", config = "", CENTER, COLOR = BLUE BOLD;
lab6 = formonly.lab6, widget = "label", config = "", CENTER, COLOR = BLUE BOLD;
lab7 = formonly.lab7, widget="label", config="", COLOR = BLUE BOLD, RIGHT;
lab8 = formonly.lab8, widget="label", config="", COLOR = BLUE BOLD, RIGHT;
lab9 = formonly.lab9, widget="label", config="", COLOR = BLUE BOLD, RIGHT;
lab10 = formonly.lab10, widget="label", config = "", LEFT, COLOR = BLUE BOLD;
lab11 = formonly.lab11, widget = "label", config = "", CENTER, COLOR = BLUE BOLD;
lab12 = formonly.lab12, widget = "label", config = "", CENTER, COLOR = BLUE BOLD;
field1 = formonly.field1, CENTER, COLOR = BOLD;
field2 = formonly.field2, CENTER, COLOR = BOLD;
field3 = formonly.field3, CENTER, COLOR = BOLD;
field4 = formonly.field4, CENTER, COLOR = BOLD;
field5 = formonly.field5, CENTER, COLOR = BOLD;
fld1 = formonly.fld1, CENTER, COLOR = BOLD;
fld2 = formonly.fld2, CENTER, COLOR = BOLD;
fld3 = formonly.fld3, CENTER, COLOR = BOLD;
fld4 = formonly.fld4, CENTER, COLOR = BOLD;
fld5 = formonly.fld5, CENTER, COLOR = BOLD;
# The rest of the list in this combo box is populated at runtime
fetch_row = formonly.fetch_row TYPE CHAR, widget="combo",
           include = ("FETCH FIRST", "FETCH LAST", "FETCH ABSOLUTE 5");
f_row_but = formonly.f_row_but, widget = "button", config = "F1 {Fetch}",
            CENTER, COLOR = GREEN BOLD;
row_id = formonly.row_id, widget="label", config = "", LEFT, COLOR = GREEN BOLD;
max_min_but = formonly.max_min_but, widget="button", config = "Max_min {}",
              CENTER, COLOR = GREEN BOLD ;
avg_sum_but = formonly.avg_sum_but, widget="button", config = "Avg_sum {}",
              CENTER, COLOR = GREEN BOLD;
count_but = formonly.count_but, widget="button", config = "Count {}",
            CENTER, COLOR = GREEN BOLD;
max_min = formonly.max_min, CENTER, COLOR = GREEN BOLD;
avg_sum = formonly.avg_sum, CENTER, COLOR = GREEN BOLD;
count_f = formonly.count_f, CENTER, COLOR = GREEN BOLD;
sel_agr = formonly.sel_agr, widget = "combo", include = ("Case 1", "Case 2",
"Case 3", "Case 4"),
          COLOR = BOLD;
aggr_but = formonly.aggr_but, widget = "button", config = "F1 {Select}",
            COLOR = GREEN BOLD;
quit_b = formonly.quit_b, widget="button", config = "F2 {Quit}",
          COLOR = RED REVERSE, CENTER;
insert_b = formonly.insert_b, widget="button", config = "insert {Insert}",
            COLOR = GREEN BOLD, CENTER;
```

INSTRUCTIONS

DELIMITERS "[]"

```
SCREEN RECORD s_rec[10] (field1 THRU field5)
SCREEN RECORD s_rec_fetch (fld1 THRU fld5)
```



Prepared Statements

4GL is a language that supports using the statements that belong to other language, SQL statements, in particular. Some of these statements can be embedded to the source code just as the usual 4GL statements, they have been discussed earlier in this manual (i.e., CREATE TABLE, ALTER TABLE, INSERT, etc.). Some of the SQL statements cannot be specified this way. They need to be prepared by means of certain statements and keywords before they can be used by a 4GL program. The embedded statements can also be prepared sometimes for convenience and rational space usage.

In this chapter, we will discuss the statements that are to be prepared, how they are to be specified and executed, as well as how the results of these executions are to be processed.

An SQL statement which cannot be embedded into 4GL can be prepared using the PREPARE statement.

The EXECUTE statement is designed to pass the prepared statements to the database server in order for it to execute them. The PREPARE statements do not execute the prepared structures, they only declare them and get them ready to be used by other statements.

When a statement is prepared, the program allocates memory space for it. Several statements and cursors can be in prepared state at one moment of time, their number depends on your system resources. You can release the resources occupied by a prepared statement by means of the FREE statement, and use the freed resources to prepare another statement or declare a cursor.

The PREPARE Statement

The PREPARE statement is used to validate, analyze, and create SQL statements execution plans at runtime. The statement refers to the text of an SQL statement during the runtime, assembles it, and makes it executable. One prepared structure can include several SQL statements. The PREPARE statement is created and applied in three steps:

- The PREPARE statement treats the specified SQL statement text as input, which can be represented by a quoted string or by a character variable containing such string. This text can include question marks which represent data values that are to be specified by the user or by the program during the statement execution.
- The OPEN and EXECUTE statements can supply the missing values if needed and execute the prepared structure once or a number of times.
- When you do not plan to use the prepared structure further in the program, you can use the FREE statement to release the resources allocated for it.

The syntax of the PREPARE statement is as follows:

```
PREPARE name FROM prepared_text [, prepared_text]
```

The *name* stands for an identifier, by which the program will afterwards refer to the prepared SQL statement. This identifier must be unique among the names of cursors and other prepared statements.

The *prepared_text* stands for a quoted string which contains either a part or the whole text of one or several SQL statements to be prepared. It can also be represented by a character variable containing such text.



The number of the statements and cursors that you can prepare is limited only by your computer memory resources.

Most of the SQL statements you already know can be prepared. But typically the [SELECT](#) or [INSERT](#) statements are prepared and used with the cursor afterwards. For example, to prepare an INSERT statement you need to assign the text of the INSERT statement to the prepared statement identifier as follows:

```
PREPARE my_prep_stmt FROM "INSERT INTO my_tab VALUES('val1', 2,  
"val3")"
```

As you can see the syntax of the INSERT statement is the same as usual and from now on the "my_prep_stmt" identified can be used instead of this INSERT statement wherever you want to use it. However, the text of the prepared statements cannot include 4GL variables, thus in our case the INSERT statement will be able to insert only literal values. There are ways to supply the values and parameters for the prepared INSERT and SELECT statements at runtime to make them more dynamic. These methods are discussed later in this chapter.

The text of the insert statement is enclosed in quotation marks. The example below has the same effect, but a character variable is used instead of the quoted string in the FROM clause:

```
LET prep_text = "INSERT INTO my_tab VALUES('val1', 2, "val3")"  
PREPARE my_prep_stmt FROM prep_text
```



Note: The text of the prepared statement cannot contain a 4GL variable. If an SQL statement cannot be embedded and requires host variables specifying input and output options, you should use the question mark placeholders which are discussed later.

The Name of a Prepared Statement

The prepared statement is intended to send a statement text to the database server which analyzes it and converts to an internal form, if no syntax errors are detected. The translated statement is saved in a data structure allocated by the PREPARE statement and can be executed in future. The prepared statement identifier specifies the name of the data structure, which contains the prepared statement. Other SQL statements can use the statement identifiers to reference the prepared statements.

The default scope of reference of a prepared statement is the module where it was declared. You cannot reference the identifier of a statement in one module, if it was prepared in another one. If you need to reference a statement identifier from outside the module where it was declared, you have to use a variable of the PREPARED data type to operate a prepared statement.



	<p>Note: Some database server support statement identifiers which are not longer than nine characters.</p>
---	---

The FREE statement followed by the name of a prepared statement releases the resources that the program has allocated to this prepared statement. After the FREE statement is executed, the statement identifier specified in it cannot be referenced by an EXECUTE statement or by a cursor, until you link the same or another statement to the identifier once more.

A prepared statement name may refer to only one prepared statement at a time. You can assign the text of another statement you want to prepare to an existing prepared statement identifier using the PREPARE statement again with the same statement name.

The PREPARED Data Type

Another way of preparing SQL statements for execution is to declare a variable of the object data type PREPARED:

```
DEFINE variable_name PREPARED
```

variable_name here stands for an identifier which will be referenced by the 4GL code. It must be unique among all the other variable names in the program. The word PREPARED designates the data type for the variable.

In the following line of code a variable of the PREPARED data type with the name *prep* is declared:

```
DEFINE prep PREPARED
```

A variable defined this way can be used in conjunction with specific methods to prepare SQL statements as explained further in this chapter.

The scope of REFERENCE of a PREPARED variable can be local, module or global. That is, if a variable of the PREPARED data type was defined within a function, for example, it will be visible only there.



Text of the Prepared Statement

As we have noticed before, the statement text can be represented by a quoted string or by a character variable containing such string. When you specify the statement to be prepared in the FROM clause, you have to keep to the following rules:

- The statement text can contain only SQL statements. You cannot include 4GL, C, or C++ statements to the statement text. There is also a list of SQL statements that cannot be prepared (they are described below in this chapter).
- The statement text can include comments enclosed in braces ({}) or preceded by two hyphens. The rules of the usage of these comments indicators are those which are in effect for other comments. You cannot use the pound symbol (#) within the PREPARE statement as a comment indicator. E.g.:

```
PREPARE prep_stmt FROM "SELECT * {comment} FROM my_tab"  
PREPARE prep_stmt1 FROM "SELECT * -- comment  
                           FROM my_tab"
```

- Database elements (e.g., column and table names) are the only valid identifiers that can be used within the prepared SQL statements. It is impossible to prepare a SELECT statement containing an INTO clause which requires a 4GL variable or an INSERT statement which VALUES clause contains a 4GL variable.
- If you need your prepared statement to be dynamic you can specify the place to where the value should be supplied at runtime. This place is indicated by a question mark (?) which is called a placeholder. It can be used instead of 4GL variables in the above mentioned cases. An SQL identifier cannot be supplied to this place. E.g.:

```
PREPARE prep_stmt2 FROM "INSERT INTO my_tab VALUES (?, ?, ?)"
```

Preparing a SELECT Statement

A prepared SELECT statement **must not** include the INTO clause. Consequently, the INTO TEMP clause is also invalid, if the statement is prepared.

If you need to use a prepared SELECT statement, you must use it with a cursor. To use a prepared statement with a cursor you just need to specify the name of the prepared statement in the FOR clause of the DECLARE statement. The variables to receive the query results should be supplied by the INTO clause of the [FETCH](#) or [FOREACH](#) statements processing the corresponding cursor.

Here is an example of a prepared SELECT statement used with the cursor where the INTO clause is placed in the FOREACH statement:

```
PREPARE my_prep FROM "SELECT * FROM custom"  
  
DECLARE cur CURSOR FOR my_prep -- declares a cursor for the  
selected  
                           -- statement  
OPEN cur  
FOREACH cur INTO pers.*      -- here is the INTO clause for the  
query  
                           -- results  
...  
END FOREACH
```



```
CLOSE cur
```

A prepared SELECT statement can include a FOR UPDATE clause which is used to declare an update cursor together with the DECLARE statement. In the following example, a SELECT statement includes a FOR UPDATE clause:

```
LET sel_stmt = "SELECT * FROM custom FOR UPDATE OF age"  
PREPARE upd_sel FROM sel_stmt
```

Preparing a SELECT Statement Associated with a Variable of the PREPARED Data Type

A SELECT statement can also be prepared by means of the *Prepare()* method called on a variable of the PREPARED data type.

```
CALL variable_name.Prepare(SELECT_statement[, isNative])
```

Here *variable_name* stands for the identifier of the PREPARED data type variable you declared earlier in your program. The *Prepare()* method, called on this variable, is joined to it by means of the dot operator and takes two arguments.

The first argument is obligatory. It is an SQL statement which can be represented either by a quoted character string or a variable of a character data type which stores this string.

The second argument – *isNative* – is optional. It is used to specify whether the SELECT statement should be changed by the SQL translator into a form compatible with other databases such as Oracle or MSSQL Server, or be kept as it is for working with the Informix databases only.

isNative accepts a value of the BOOLEAN data type, and can be set either to FALSE (by default) to bypass the SQL translator or TRUE if the translation is required.

The following code prepares a SELECT statement and requests the translation of this statement into a form compatible with other databases.

```
CALL prep.Prepare("SELECT * FROM tab_1 WHERE id = 3", TRUE)
```



Preparing Statements with Known and Unknown Parameters

We have mentioned before that 4GL provides a programmer and a user with an ability to pass values to the prepared statements both during the statement preparation and at runtime.

Preparing Statements with Known Parameters

Sometimes the parameters that are to be applied to a prepared statement are already known during the statement specification. These parameters can be included to the prepared statement text and cannot be changed at runtime.

The statements that are the most often prepared are the SELECT and the INSERT statements. The syntax of the SELECT statement, if it is prepared, slightly differs from the syntax of the embedded SELECT statement and the difference is described in the section above.

By the parameters of the SELECT statement we understand the WHERE clause which contains the query conditions. The only identifiers allowed in a prepared statement are the SQL identifiers (the names of tables, columns, etc.). Thus the WHERE clause can contain SQL identifiers and literal values, if you prepare a SELECT statement with known parameters. Here is an example of a prepared SELECT statement:

```
PREPARE cus_id
    FROM "SELECT name      -- "name" is an SQL identifier (column name)
          FROM custom     -- "custom" is an SQL identifier (table name)
          WHERE id > 231" -- here "id" is an SQL identifier and
                            -- 231 is a literal number

DECLARE cur CURSOR FOR cus_id
OPEN cur
FETCH cur INTO pers.*
...
CLOSE cur
```

The prepared INSERT statement has the same syntax as the embedded one. If you prepare an INSERT statement, you have to create a cursor for it and use the [PUT](#) statement in order to send the specified values to the database table. The prepared INSERT statement must include the VALUES clause, but as no 4GL variables are allowed, you can use only literal values in it, if you prepare the INSERT statement which parameters are supplied by the programmer.

```
PREPARE cus_addr
    FROM "INSERT INTO addr
          VALUES
          (003, 'New York', 'Grand av.', 345)" --all values are literals

DECLARE inc CURSOR FOR cus_addr -- we declare the insert cursor

OPEN inc

PUT inc

CLOSE inc -- inserts a single row of specified values
```



Preparing Statements with known Parameters Using a PREPARED Variable

The same rule about using only SQL identifiers or literal values as known parameters in a prepared statement applies to a statement prepared with the help of a variable of the PREPARED data type.

A prepared SELECT statement in this case also must be used with a cursor in order to retrieve multiple rows. After defining a PREPARED variable and specifying an SQL query in the *Prepare()* method, you should call the *Declare()* method on a CURSOR variable and pass the PREPARED variable as its argument. Then the cursor will process the prepared statement as described in the chapter on cursors.

For example:

```
DEFINE prep PREPARED, cur CURSOR

# The parameter is known here. It is a literal integer

CALL prep.Prepare ("SELECT * FROM tab_1 WHERE price > 50")

# We call a Declare() method on a CURSOR variable passing the PREPARED
# variable as the argument

CALL cur.Declare(prep)
...
```

As for a prepared INSERT statement with known parameters, you should call the *Put()* and *Flush()* or *Close()* methods on a variable of the CURSOR data type in order to insert values specified in the SQL statement into the database table from the insert buffer.

```
DEFINE prep PREPARED, cur CURSOR

CALL prep.Prepare("INSERT INTO tab_1 values (2, "William")")

CALL cur.Declare(prep)

CALL cur.Open()

CALL cur.Put() -- The Put() method will insert the values from the INSERT
               -- statement into the insert buffer
CALL cur.Flush()--And the Flush() method will put the values from the
               -- insert buffer into the table

...
```



Preparing Statements with Parameters Received at Runtime

Sometimes, when the statements are prepared, their parameters may be unknown to the programmer, because they may vary each time the statement is executed. If this is the case, you can use question marks as placeholders for the parameters that are to be supplied during the execution. A prepared statement containing such placeholders should be used with somewhat modified statements which are discussed later in this chapter.

The following examples demonstrate how a placeholder can be used in the prepared statement declaration:

```
PREPARE my_s FROM "SELECT * FROM addr WHERE zipcode = ?"  
PREPARE my_2 FROM "INSERT INTO first_tab VALUES ('test', ?, ?)"
```

An SQL identifier (database name, table name, column name) cannot be represented by a question mark.

The prepared statements with placeholders are processed in a slightly different way than the prepared statements with known values. It is discussed in details later in this chapter.

The same can be done using the PREPARE data type. You can also set the placeholders instead of the values here, e.g.:

```
CALL prep_v.Prepare("INSERT INTO first_tab VALUES ('test', ?, ?)")
```

Preparing Statements with Unknown SQL Identifiers

Question mark placeholders cannot replace table or column names; these items have to be specified in the statement text within the PREPARE statement.

However, 4GL provides a way to pass SQL identifiers from the user input, if they are not available during the statement declaration. In the following example, the user is to enter the names of the columns to be queried. The input names are sent to the character data type variable, which is finally specified as a statement text identifier within the PREPARE statement.

```
DEFINE col_name varchar (40),  
      sel_stm varchar(100),  
      col1,col2,col3,col4 varchar (20)  
  
PROMPT "Enter the list of the columns to be queried " FOR col_name  
  
LET sel_stm = "SELECT ", col_name, " FROM custom"  
DISPLAY sel_stm at 1,1  
SLEEP 2  
PREPARE prs FROM sel_stm  
  
DECLARE curl CURSOR FOR prs  
OPEN curl  
FOREACH curl INTO col1,col2,col3,col4  
DISPLAY col1, col2  
END FOREACH  
CLOSE curl
```



Note: When a 4GL compiler compiles the source code which contains prepared statements, it doesn't check them for syntax errors. If the prepared statement text violates the syntax, the error will be detected only at runtime.

Referencing Prepared Statements With Placeholders

The section below dwells upon the specific requirements which should be met when referencing a prepared statement with placeholders.

Referencing a SELECT Statement with Placeholders

You cannot process the cursor associated with the SELECT statement using the FOREACH statement, if the SELECT statement contains a question mark placeholder. In this case, OPEN, FETCH, and CLOSE statements can prove useful. However, the OPEN statement must include an optional clause which is required only in this case. The syntax of the OPEN statement used for processing the SELECT statement with placeholders is as follows:

```
OPEN cursor_name USING variable, [, variable...]
```

Here the cursor name is the name of a cursor associated with a prepared SELECT statement which contains at least one placeholder. The variable list must include one or more 4GL variables which will supply the values for the placeholders. The number of placeholders and their order must coincide with the number and order of the placeholders in the prepared statement.

In the example below the prepared SELECT statement contains two placeholders. The values supplied instead of these placeholders are entered by the user at runtime:

```
PREPARE prep_stmt FROM "SELECT * FROM custom WHERE age_col < ? AND
age_col > ?"      -- the prepared statement has 2 placeholders

# We receive query parameters from the user
PROMPT "Enter the maximum age?" FOR age_max
PROMPT "Enter the minimum age?" FOR age_min

DECLARE cur CURSOR FOR prep_stmt

OPEN cur USING age_max, age_min      -- opens the cursor and sends a
values                                -- to the unknown parameters
WHILE TRUE
    FETCH cur INTO pers.*
    ...
END WHILE
CLOSE cur
```



Referencing a SELECT Statement with Placeholders when Using a PREPARED Variable

To process a SELECT statement with placeholders using a variable of the PREPARED data type you should follow different syntax, which is illustrated by the following code fragment – a modified version of the example above.

You need to use the SetParameters() method to specify the values for the placeholders. This method accepts the number of arguments equal to the number of placeholders used in the SELECT statement. You can call this method for the PREPARED variable or for the CURSOR variable, for which the PREPARED variable is used. In the example below it is used for the CURSOR variable after the cursor is declared using the Declare() method.

	Note: The SetParameters() method must be called before the cursor is opened, if it is required.
---	--

The SetResults() method specifies the output for the prepared statement. It accepts the number of parameters which corresponds to the number of returned values. The arguments can be represented by variables of simple or structured data types. This method functions in the same way as the INTO clause of the FETCH or FOREACH statement.

```
DEFINE age_maximum,
      age_minimum
      prep
      cur
                  INT,
                  PREPARED, -- A PREPARED variable is defined
                  CURSOR,   -- A CURSOR variable is defined

# We receive query parameters from the user too
      PROMPT "Enter the maximum age?" FOR age_max
      PROMPT "Enter the minimum age?" FOR age_min

# Then the Prepare() method with the SELECT statement is called on
# the
# PREPARED variable prep

CALL prep.Prepare("SELECT * FROM custom WHERE age_col<? AND age_col>?")

CALL cur.Declare (prep)--The PREPARED variable is sent as the
argument to
                           --the Declare()method

# Then we specify the variables which will receive input from the user
# and which are represented by placeholders in the SELECT statement by
# calling the SetParameters() method. With a SELECT statement, this
# method must be called prior to opening the cursor

CALL cur.SetParameters (age_max, age_min)

# The results retrieved by the cursor will be stored in the
variables
```



```
# age_maximum and age_minimum

CALL cur.SetResults(age_maximum,age_minimum)
CALL cur.Open()
WHILE (cur.FetchNext() = 0)
    CALL cur.FetchNext()
    DISPLAY age_minimum, "", age_maximum
END WHILE
CALL cur.Close()
CALL cur.Free()
```

Referencing an INSERT Statement with Placeholders

To process a cursor associated with a prepared INSERT statement which contains placeholders you do not need a special case of the OPEN statement. However, you need a special clause added to the PUT statement processing such cursor. The syntax of the PUT statement dealing with the placeholders is as follows:

```
PUT cursor_name FROM variable [, variable]
```

Here the cursor name is the name of the cursor associated with a prepared INSERT statement containing placeholders. The variable list must include one or more 4GL variables which will supply the values for the placeholders. The number of placeholders and their number must coincide with the number and order of the placeholders in the prepared statement.

The following example illustrates how the question mark placeholder can be used within a prepared INSERT statement. If a prepared INSERT statement has a question mark placeholder, the source of the data to be inserted is specified in the FROM clause of the PUT statement:

```
PREPARE ins FROM "INSERT INTO goods VALUES (?, ?, ?, ?)"

DECLARE ins_cur CURSOR
    FOR ins -- declares cursor for the INSERT statement

OPEN ins_cur

FOR a = 1 to 5

#We specify values which will replace the placeholders

LET art.* = a, "Item", "Provider", article+1

PUT good FROM art.* -- we reference the replacement values for
-- placeholders in the FROM clause

END FOR

CLOSE inc
```



Referencing an INSERT Statement with Placeholders by Means of a PREPARED Variable

With a PREPARED variable you must define a variable of the CURSOR data type and use the *Put()* and *Flush()* or *Close()* methods on it to insert the values represented by placeholders in the INSERT statement into a database table.

The *SetParameters()* method is used to specify the values for the placeholders. In case of the insert cursor this method can be called either before the *Open()* method or after the *Open()* method, unlike the select cursor.

```
DEFINE prep PREPARED, cur CURSOR, art RECORD LIKE tab_1.art.* , a INT
CALL prep.Prepare("INSERT INTO tab_1 VALUES (?, ?, ?, ?, ?)")

CALL cur.Declare (prep)
CALL cur.setParameters(art.*)

CALL cur.Open()

FOR a = 1 to 5
    LET art.* = a, "Item", "Provider", article+1
    CALL cur.Put()
END FOR

CALL cur.Close()
```

The EXECUTE Statement

The EXECUTE statement passes a previously prepared statement or a set of statements to the database server which executes them, if no errors are detected. This is another way to execute the prepared statements along with associating them with cursors. If the passed statement has one or more question mark placeholders, they are supplied with specific values before the execution. When an SQL statement becomes prepared, it can be executed as often and as many times as it is necessary, until the FREE statement is performed.

The syntax of the EXECUTE statement is as follows:

```
EXECUTE prepared_stmt [INTO clause] [USING clause]
```

The *prepared_stmt* stands for the prepared statement identifier that must be defined in the PREPARE statement earlier in the source code text. It can also be a variable which contains the name of the prepared statement. The EXECUTE statement or the DECLARE cursor statement cannot refer to the statement



identifier, if the program has already used the FREE statement to release the database server resources allocated to the specified prepared statement.

The EXECUTE statement can be applied to any prepared statement. However, if a prepared SELECT statement returns multiple rows, the data rows can be retrieved by means of the DECLARE, OPEN, and FETSCH cursor statements.

The simple example of the EXECUTE statement application is given below:

```
PREPARE sel FROM "INSERT INTO goods VALUES ("string", 100)"  
EXECUTE sel
```

The INTO Clause

The INTO clause is used to make it possible for the EXECUTE statement to process a SELECT statement retrieving exactly one row and store the resulting values into output variables. While no INTO clause is allowed in the prepared SELECT statement, the INTO clause of the EXECUTE statement may come in handy.

The generalized syntax of the INTO clause of the EXECUTE statement is the following:

```
INTO output_variable [, output_variable...]
```

The *output variable* is a variable which is to receive the value retrieved by the SELECT statement.

If the INTO clause is specified for a prepared SELECT statement which returns multiple rows, an error occurs. To be able to use such statement you need to associate it with a cursor.

The output variable cannot receive a null value. If you know that one of the referenced columns may contain a null value, perform a check of the variable linked to the column to determine, if the received value is null.

You have to perform the following steps to use the EXECUTE statement with the INTO clause:

- Declare an output variable (or variables) that will be used by the INTO clause;
- Prepare the SELECT statement by means of the PREPARE statement;
- Execute your SELECT statement by means of the EXECUTE statement with an INTO clause

Here is a simple example of the INTO clause application:

```
DEFINE pers RECORD  
    id int,  
    f_name,l_name CHAR (20),  
    age int  
END RECORD  
  
PREPARE sel FROM "SELECT * FROM custom WHERE ID = 15"  
EXECUTE sel INTO pers.*
```



The USING Clause

The USING clause is used to specify the values that are to replace question marks placeholders, if they are used within the prepared statement. The process of supplying the prepared statement with values instead of question mark placeholders is also called the *parametrizing* of the prepared statement.

The syntax of the USING clause is as follows:

```
USING (variable [, variable...])
```

The *variable name* is a host variable identifier. The value of this variable replaces the question mark placeholder before the program executes the prepared statement. The variables in this case can be substituted by literal values.

Using the host variable is the most common way of supplying the values to the prepared statements. Each placeholder should have a corresponding variable which will supply it with data. The value stored in the host variable must be compatible with the corresponding value required by the prepared statement.

```
PREPARE my_ins FROM "INSERT INTO custom VALUES (?, ?, ?, ?, ?)"  
EXECUTE my_ins USING (3, "Rickie", "Alisson", 27)
```

Executing a Prepared Statement Associated with a Variable of the PREPARED Data Type

If a statement was prepared with the use of a variable of the PREPARED data type, to execute it you should call the *Execute()* method.

```
CALL variable_name.Execute()
```

This method takes no parameters and returns a value of the sqlca.sqlcode record member.

The code below first prepares an SQL statement and then passes it to the database server by calling the *Execute()* method.

This method does not have the functionality of the USING and INTO clause of the EXECUTE statement. To specify the output destination for the prepared statement the SetResults() method is used along with the Execute method. To specify the values to replace the placeholders, use the SetParameters() method together with the Execute() method.

```
...  
DEFINE prep PREPARED  
  
CALL prep.Prepare("SELECT * FROM tab_1 WHERE id > ?")  
CALL prep.SetParameters(100)  
CALL prep.SetResults(select_rec.*)  
CALL prep.Execute()  
...
```



The code above will have the same result as the code below which uses 4GL statements instead of methods:

```
...  
    PREPARE prep_st FROM "SELECT * FROM tab_1 WHERE id > ?"  
    EXECUTE prep_stmt INTO select_rec.* USING (100)  
...  
...
```

The FREE Statement

We have already discussed the FREE statement in regard to freeing the resources allocated to a cursor. The same statement can be used to release resources that are allocated to a prepared statement in a database server and in the used application-development tool. The resources are allocated at the moment when a statement is prepared or a cursor is opened, and remain unavailable for other items until they are released by the FREE statement.

The syntax of the FREE statement is very simple:

```
FREE prepared_stmt_name
```

If the FREE statement is applied to a prepared statement which is not associated with a cursor, it frees the resources in both the database server and application development tool. If a prepared statement is associated with a cursor, the structure like *FREE statement_id* can release only the resources of the application development tool. The database server resources can be freed only if the cursor is freed.

If a statement or a cursor were specified in a FREE statement, they can be executed again only after they are prepared (a statement) or declared (a cursor) once more.

Freeing Resources Allocated to a Statement Prepared with a PREPARED Variable

To free resources associated with a statement prepared by defining a PREPARED variable, the *Free()* method is used.

```
CALL variable_name.Free()
```

The method takes no arguments and returns a value of sqlca.sqlcode.

After the *Free()* method is executed you cannot reference the statement you prepared with a variable of the PREPARED data type, and you will have to prepare it again by means of the *Prepare()* method.

Preparing Sequences of Multiple SQL Statements

Several SQL statements can be executed in a row, if you prepare them in a single prepared statement. To associate several statements with one prepared statement identifier, you need to include them into the prepared text.



However, a prepared multi-statement is processed as a unit. All the statements comprising it are executed simultaneously and not one after another. This means that you cannot prepare a multi-statement in which the execution of the next statement depends on the result of the execution of the previous one. Thus you can prepare several SELECT statements at once, but you cannot prepare the CREATE TABLE and INSERT statement inserting values into that table using one PREPARE statement.

To prepare a multiple statement you need enclose each following statement in quotation marks and separate them by commas. However, you also need to add a semicolon at the end of each statement included into the multiple statement. Here is an example of a multiple statement consisting of several UPDATE statements:

```
PREPARE upd_prep
    FROM "UPDATE cust SET age = ? WHERE lname = ? ;",
         "UPDATE cust SET age = ? WHERE lname = ? ;",
         "UPDATE cust SET age = ? WHERE lname = ? ;"

EXECUTE upd_prep USING age1, lname1, age2, lname2, age3, lname3
```

In the example above the prepared multi-statement contains three UPDATE statements with placeholders. The USING clause of the EXECUTE statement contains 6 variables which supply values for these placeholders in the order of their appearance.

Statements that Can, Must, and Must not be Prepared

You can prepare most SQL statements. You even *must* prepare some of the SQL statements, if you want to use them in a 4GL program, because they cannot be embedded.

Statements that Must be Prepared

Some of the statements cannot be directly embedded to the program, so, they must be prepared by the PREPARE statement. These statements are as follows:

ALTER FRAGMENT	SET CONSTRAINT
ALTER OPTICAL CLUSTER	SET DATABASE OBJECT MODE
CREATE EXTERNAL TABLE	SET DATASKIP
CREATE OPTICAL CLUSTER	SET DEBUG FILE TO
CREATE ROLE	SET LOG
CREATE SCHEMA	SET MOUNTING TIMEOUT
CREATE TRIGGER	SET OPTIMIZATION
DROP OPTICAL CLUSTER	SET PDQPRIORITy
DROP PROCEDURE	SET PLOAD FILE
DROP ROLE	SET PRESIDENCY
DROP TRIGGER	SET ROLE
EXECUTE PROCEDURE	SET SCHEDULE LEVEL
GRANT FRAGMENT	SET SESSION AUTORIZATION
RELEASE	SET TRANSACTION
RENAME DATABASE	SET TRANSACTION MODE
RESERVE	SET VIOLATIONS TABLE
REVOKE FRAGMENT	STOP VIOLATIONS TABLE

For more information about these statements, please, refer to the SQL syntax guide, as they are out of the scope of this manual.



Statements that Cannot be Prepared

4GL also supports a set of SQL statements that can be embedded to the program directly but cannot be prepared. Here is the list of such statements:

CLOSE	FREE
CONNECT	LOAD
DECLARE	OPEN
DISCONNECT	PREPARE
EXECUTE	PUT
FETCH	UNLOAD
FLUSH	WHENEVER

These statements have been described before in this manual, they can be directly embedded into the 4GL source code.

SQL... END SQL

There is an alternative way of preparing SQL statements in 4GL. It is to use the SQL ... END SQL delimiters.

```
SQL
  SQL statement
END SQL
```

When you place an SQL statement within these delimiters, it will be automatically prepared, executed and freed, which facilitates the preparation process.

In the following code snippet first a scroll cursor is declared. Then a SELECT statement is prepared by means of the SQL ... END SQL block:

```
DECLARE c_sel_clients_2 SCROLL CURSOR FOR

SQL
  SELECT id_client,first_name||last_name FROM clients
END SQL

OPEN c_sel_clients_2

FETCH NEXT c_sel_clients_2 INTO r_clients.id_client, full_name

CLOSE c_sel_clients_2

DISPLAY r_clients.id_client USING "###", "
",full_name,"",r_clients.start_date," ",r_clients.end_date AT 2,2
```

You should be aware of some peculiarities of the SQL ... END SQL block usage in comparison with the PREPARE statement. First, only one SQL statement can be placed within the delimiters like in the example



above. Besides, if you want to use the concatenation operator “||” to combine values from several columns into one, as in the fragment of the same code given below, it can be done only within the SQL ... END SQL delimiters in contrast to the PREPARE statement where this is impossible.

```
SQL
  SELECT id_client,first_name||last_name FROM clients
END SQL
```

Here the values from the columns “first_name” and “last_name” are combined into one value.

Example

This application includes the PREPARED statement used with and without cursor as well as the PREPARED data type used with a cursor and independently.

```
#####
# This example uses dynamic SQL with the statements PREPARE, EXECUTE, OPEN
# and FREE. It also uses the CURSOR and PREPARED data types
#####

DATABASE db_examples

MAIN

MENU "Preparing"

COMMAND "PREPARED statement"
  MENU "Statement"

    COMMAND "Simple PREPARE"
      "Declaring a cursor for a prepared statement"
      CALL prep_simple()

    COMMAND "PREPARE with Placeholders"
      "Preparing a statement with placeholders for a cursor"
      CALL prep_placeholder()

    COMMAND "EXECUTE"
      "Executing a prepared statement with the EXECUTE statement"
      CALL prep_execute()

    COMMAND "Multiple Statements"
      "Preparing several SQL statements into one prepared statement"
      CALL prep_multiple()

    COMMAND "Return"
      EXIT MENU

  END MENU

COMMAND "PREPARED datatype"

  MENU "Datatype"
    COMMAND "With a Cursor"
      "The PREPARED variable is associated with a CURSOR variable"
```



```
        CALL prep_var_cursor()
COMMAND "Independent"
        "The PREPARED variable is not associated with a cursor"
        CALL prep_var_indep()
COMMAND "Return"
        EXIT MENU
END MENU

COMMAND "Complex Usage"
        "Using prepared statements for inserting, updating and deleting"
        CALL prep_complex()
COMMAND "Exit"
        EXIT PROGRAM
END MENU

END MAIN

#####
# This function illustrates a simple case of a prepared
# statement associated with a read-only select cursor
#####
FUNCTION prep_simple()
DEFINE
    r_clients      RECORD LIKE clients.*,
    full_name      CHAR(40),
    stmt           CHAR(512),
    v_date_1       DATE,
    i_row,i        INTEGER

CLEAR SCREEN

DISPLAY "Using the PREPARE statement and cursor with OPEN, FETCH, CLOSE."
AT 3,2 ATTRIBUTE(GREEN, BOLD)

LET v_date_1 = "09/10/2010" -- we assign the date to be used

# Then we assign the statement text to a variable - all the operands are glued
# into one character string

LET stmt = "SELECT c.* FROM clients c WHERE c.start_date = """",
        -- a part of the SELECT statement as a character string
    v_date_1,   -- we use the variable to specify the part of the
        -- WHERE condition
    """ ",     -- the variable is surrounded by the literal
        -- quotation marks,
        -- because its value needs to be
        -- enclosed in them according to the SELECT syntax
    "ORDER BY c.id_client" -- then we assign the rest
        -- of the statement as a string

PREPARE p_stmt_1 FROM stmt -- after that we use the variable
                           -- with the text in the PREPARE statement

DECLARE c_stmt_1 CURSOR FOR p_stmt_1 -- and we declare the cursor
```



```

-- for the prepared SELECT statement

# Then we can use the cursor declared for the prepared statement in a normal way
# And the result is the same as if we used a cursor for a non-prepared statement

OPEN c_stmt_1

LET i_row = 4 -- we use this variable as the display coordinate
WHILE TRUE
    FETCH c_stmt_1 INTO r_clients.* -- we use this cursor to
                                    -- retrieve and display values
    IF status = NOTFOUND THEN EXIT WHILE END IF
    DISPLAY r_clients.id_client USING "###", "", r_clients.first_name,
        r_clients.last_name, " ", r_clients.start_date, " ",
        r_clients.end_date AT i_row, 2
    LET i_row = i_row + 1
    IF i_row > 23
        THEN LET i_row = 4
        CLEAR SCREEN
        DISPLAY "Using the PREPARE statement with OPEN, FETCH, CLOSE and FREE"
            AT 3,2 ATTRIBUTE(GREEN, BOLD)
    END IF
END WHILE
CLOSE c_stmt_1
FREE c_stmt_1 -- the FREE statement releases the declared
              -- cursor, it cannot be used any more
END FUNCTION

#####
# This function prepares an SQL statement with placeholders
# and associates it with a cursor. Then this cursor is processed
# using the FOREACH statement with the USING clause
#####
FUNCTION prep_placeholder()
DEFINE
    r_clients      RECORD LIKE clients.*,
    stmt          CHAR(512),
    v_date_1       DATE,
    v_date_2       DATE,
    i_row, i       INTEGER

    CLEAR SCREEN

    DISPLAY "Using the PREPARE statement with placeholders: " AT 3,2
    ATTRIBUTE(GREEN, BOLD)
    # The text of the SELECT statement to be prepared is contained in a character
    # string which is used directly in the PREPARE statement and is not assigned to
    # a variable
    # The character string contains the place holders (?) instead of the values for
    # v_date_1 and v_date_2.

    PREPARE p_stmt_2 FROM "SELECT id_client,first_name,last_name, date_birth, ||"
                        "end_date FROM clients WHERE date_birth >= ? and " ||
                        "end_date = ? ORDER BY date_birth"

```



```

# The cursor is declared for this statement in the usual way

DECLARE c_stmt_2 CURSOR FOR p_stmt_2

# Here we assign some values to the variables, which will replace
# the placeholders, when the cursor is used
LET i_row = 4
LET v_date_1 = DATE("01/01/1970") -- this value will be used
-- instead of the question mark
-- here: date_birth >= ?
LET v_date_2 = DATE("12/31/9999") -- this value will be used
-- instead of the question mark
-- here: end_date = ?

FOREACH c_stmt_2      -- opens the cursor for the SELECT with placeholders
    USING v_date_1,v_date_2 -- the USING clause specifies which variables
                           -- to use instead of the placeholders
    INTO r_clients.id_client, r_clients.first_name, r_clients.last_name,
          r_clients.date_birth, r_clients.end_date

    DISPLAY r_clients.id_client USING "###", " ",r_clients.first_name,
        r_clients.last_name, " ",r_clients.date_birth, " ",
        r_clients.end_date AT i_row,2
    LET i_row = i_row + 1
    IF i_row > 23
    THEN LET i_row = 4
        CLEAR SCREEN
    DISPLAY "Using the PREPARE statement with placeholders "
        AT 3,2 ATTRIBUTE(GREEN, BOLD)
    END IF
END FOREACH
FREE c_stmt_2

END FUNCTION

#####
# This function prepares a statement that retrieves one record
# and processes it using the EXECUTE statement
#####
FUNCTION prep_execute()

DEFINE
    r_clients      RECORD LIKE clients.*,
    stmt           CHAR(512),
    i              INTEGER,
    v_char_1       CHAR(80)

    CLEAR SCREEN

    DISPLAY "Executing the prepared statements with the EXECUTE statement"
        AT 1,2 ATTRIBUTE(GREEN, BOLD)

    # The EXECUTE statement executes a prepared SELECT statement returning one
    # record.
    LET stmt = "SELECT c.* FROM clients c WHERE c.id_client = 3"

```



```

PREPARE p_stmt_5 FROM stmt -- we prepare a simple SELECT
                                -- statement from a variable
INITIALIZE r_clients.* TO NULL
EXECUTE p_stmt_5 INTO r_clients.* -- and execute it
IF status = NOTFOUND
    THEN DISPLAY "Record not found" AT 3,2 ATTRIBUTE(RED)
    ELSE DISPLAY "Prepared statements without placeholders is executed:"
        AT 3,2 ATTRIBUTE(GREEN)
        DISPLAY r_clients.id_client USING "###", " ",
            r_clients.first_name,r_clients.last_name, " ",
            r_clients.start_date," ",r_clients.end_date AT 4,2
END IF

LET stmt = "SELECT c.* FROM clients c WHERE c.id_client = ?"

PREPARE p_stmt_6 FROM stmt      -- a SELECT statement with placeholders
                                -- is prepared form a variable
LET i = 5 -- we initialize the variable used instead of the placeholder

EXECUTE p_stmt_6 INTO r_clients.* -- then we execute the prepared statement
                                -- substituting the placeholder with i

IF status = NOTFOUND
    THEN DISPLAY "Record not found" AT 5,2 ATTRIBUTE (RED)
    ELSE DISPLAY "Prepared statements with placeholders is executed:"
        AT 5,2 ATTRIBUTE (GREEN)
        DISPLAY r_clients.id_client USING "###", " ",
            r_clients.first_name,r_clients.last_name, " ",
            r_clients.start_date," ",r_clients.end_date AT 6,2
END IF

END FUNCTION

#####
# This function prepares two UPDATE statements as a single
# prepared statement and executes it. It illustrates how the
# statements are executed parallelly and produce independent
# results.
#####
FUNCTION prep_multiple()

DEFINE
    r_clients      RECORD LIKE clients.*,
    r_contracts    RECORD LIKE contracts.*,
    stmt          CHAR(512),
    v_date_1       DATE,
    i              INTEGER

    CLEAR SCREEN

# We specify two UPDATE statements to be prepared in one prepared statement at
# the same time, both of them have placeholders; they update different tables
    LET stmt = "UPDATE clients SET end_date = ? WHERE id_client = ?;",
        "UPDATE contracts SET end_date = ? WHERE id_client = ?"

```



```
DISPLAY "Two UPDATE statements were prepared, but they have different outcome"
      AT 4,2 ATTRIBUTE(UNDERLINE)

PREPARE p_stmt_11 FROM stmt    -- and prepare them
LET v_date_1 = TODAY
LET i = 9      -- this value will be used in the WHERE
               -- clause of the both UPDATE statements
               -- note that this will cause the second
               -- UPDATE statement to fail
               -- because the contracts table does not contain
               -- a contract record for id_client = 9 whereas
               -- the clients table does

# As we have 4 placeholders on the whole we need 4 variables in the USING clause
# even if you want to use some values twice or more times you still
# have to specify exactly 4 variables but you may repeat them like in our case
EXECUTE p_stmt_11
      USING v_date_1, i, -- these variables will be used
                     -- for the first UPDATE statement
      v_date_1, i -- these will be used for the second one

# Selecting and displaying the row updated in clients table
# by the first UPDATE in the prepared statement
SELECT c.* INTO r_clients.* FROM clients c WHERE c.id_client = i
IF status = NOTFOUND
THEN DISPLAY "1)Record not found" AT 5,2 ATTRIBUTE(RED)
ELSE DISPLAY "1)The record has been updated:" AT 5,2 ATTRIBUTE(GREEN)
      DISPLAY r_clients.id_client USING "###", " ",
      r_clients.first_name,r_clients.last_name, " ",
      r_clients.start_date," ",r_clients.end_date AT 6,2

END IF

# Trying to select and display the row intended for update by the second
# UPDATE of the prepared statement in contracts table.
# But as there is no row where id_clients=9, the SELECT returns NOTFOUND
SELECT c.* INTO r_contracts.* FROM contracts c
      WHERE c.id_client = i AND c.account = "26302010009"

IF status = NOTFOUND
THEN DISPLAY "2)Record with id_client=9 in contracts was not found!"
      AT 7,2 ATTRIBUTE (GREEN)
ELSE DISPLAY "2)The record was found and updated"
      AT 7,2 ATTRIBUTE (RED)
      DISPLAY r_contracts.id_client USING "###", " ",
      r_contracts.num_con," ",r_contracts.account,
      " ",r_contracts.end_date AT 8,2
END IF

# Here we restore the initial values of the columns updated above
UPDATE clients SET end_date = "12/31/9999" WHERE id_client = i
UPDATE contracts SET end_date = "12/31/9999" WHERE id_client = i
```



```
END FUNCTION

#####
# This function uses a PREPARED variable to store a prepared
# statement and associates it with the CURSOR variable
#####
FUNCTION prep_var_cursor()

DEFINE
    r_clients      RECORD LIKE clients.*,
    prep_stmt      PREPARED,
    cur_var        CURSOR,
    i_row          INTEGER,
    v_date_1       DATE,
    v_date_2       DATE

CLEAR SCREEN

DISPLAY "Using the PREPARE variable with the CURSOR variable "
AT 3,2 ATTRIBUTE(GREEN, BOLD)

CALL prep_stmt.prepare("SELECT id_client,first_name,last_name, date_birth, |||
                        "end_date FROM clients WHERE date_birth >= ? and " |||
                        "end_date = ? ORDER BY date_birth")

# This value will be used instead of the question mark
# here: date_birth >= ?
LET v_date_1 = DATE("01/01/1970")

# This value will be used instead of the question mark
# here: end_date = ?
LET v_date_2 = DATE("12/31/9999")

# Here we set the variable to replace the placeholders
CALL cur_var.SetParameters(v_date_1, v_date_2)

# And here we set the variables into which the values
# retrieved by the SELECT will be written
CALL cur_var.SetResults(r_clients.id_client, r_clients.first_name,
                       r_clients.last_name, r_clients.date_birth,
                       r_clients.end_date)

CALL cur_var.declare(prep_stmt)

LET i_row = 4
CALL cur_var.open()
WHILE (cur_var.FetchNext() = 0)

    CALL cur_var.FetchNext()

    DISPLAY r_clients.id_client USING "###", " ", r_clients.first_name,
                                r_clients.last_name, " ", r_clients.date_birth, " ",
                                r_clients.end_date AT i_row,2
    LET i_row = i_row + 1
```



```
IF i_row > 23
THEN LET i_row = 4
CLEAR SCREEN
DISPLAY "Using the PREPARE variable with the CURSOR variable "
    AT 3,2 ATTRIBUTE(GREEN, BOLD)
END IF
END WHILE

CALL cur_var.Close()
CALL cur_var.Free()

END FUNCTION

#####
# This function prepares an SQL statement in a PREPARED variable
# and then executes it without the help of the cursor.
#####
FUNCTION prep_var_indep()

DEFINE
    prep_stmt1      PREPARED,
    r_clients       RECORD LIKE clients.*,
    info int

DISPLAY "A statement stored in a PREPARED variable is executed by execute()."
    AT 3,2 ATTRIBUTE(GREEN)

CALL prep_stmt1.prepare("SELECT id_client,first_name,last_name, date_birth |
    "FROM clients WHERE id_client >= ?")

# We set the variables to be written the data to for the
# PREPARED variable
CALL prep_stmt1.SetResults(r_clients.id_client, r_clients.first_name,
    r_clients.last_name, r_clients.date_birth)

# And the parameter for the placeholder
CALL prep_stmt1.SetParameters(5)

# Then we execute the prepared statement and retrieve one row
CALL prep_stmt1.execute()

DISPLAY r_clients.id_client USING "###", " ", r_clients.first_name,
    r_clients.last_name, " ", r_clients.date_birth AT 4,2

CALL prep_stmt1.free()

END FUNCTION

#####
# This function illustrates how the INSERT, UPDATE and DELETE
# statements can be prepared and used throughout the program
#####
FUNCTION prep_complex()

DEFINE
```



```

r_clients      RECORD LIKE clients.*,
stmt          CHAR(512),
i             INTEGER

CLEAR SCREEN

# First we prepare an INSERT statement without placeholders
LET stmt = "INSERT INTO clients(id_client,id_country,first_name,last_name,",
        "Address,start_date,end_date)",
        "VALUES(11,\"840\" ,\"Lycia\" ,\"Last_name\" ,\"New_address\" ,\"\" ,
        TODAY, \"\" ,\"12/31/9999\")"
PREPARE p_stmt_8 FROM stmt

EXECUTE p_stmt_8 -- then we execute this INSERT statement

# Here we check whether the row was actually inserted and display it,
# if it was.
SELECT c.* INTO r_clients.* FROM clients c WHERE c.id_client = 11
IF status = NOTFOUND
THEN DISPLAY "Record not found" AT 4,2 ATTRIBUTES (RED)
ELSE DISPLAY "A record was inserted:" AT 4,2 ATTRIBUTES (GREEN)
    DISPLAY r_clients.id_client USING "###",
            " ",r_clients.first_name,r_clients.last_name,
            " ",r_clients.start_date," ",r_clients.end_date AT 5,2
END IF

# We prepare an UPDATE statement with placeholders to update
# the rows we have just inserted
LET stmt = "UPDATE clients SET first_name = ?,last_name = ?,start_date =",
        "?,end_date = ? WHERE id_client = 11"
PREPARE p_stmt_9 FROM stmt

# Then assign values to be used instead of the placeholders
LET r_clients.first_name = "My_Lycia"
LET r_clients.last_name = "Her_Last_name"
LET r_clients.start_date = TODAY - 1 UNITS YEAR
LET r_clients.end_date = TODAY + 1 UNITS YEAR

# And then we execute the INSERT statement with the help of the
# EXECUTE statement with the USING clause
EXECUTE p_stmt_9 USING r_clients.first_name, r_clients.last_name,
r_clients.start_date, r_clients.end_date

# Here we check whether such row exists and display the updated values
SELECT c.* INTO r_clients.* FROM clients c WHERE c.id_client = 11
IF status = NOTFOUND
THEN DISPLAY "Record not found" AT 6,2 ATTRIBUTES (RED)
ELSE DISPLAY "The row has been updated, here are the new values:" AT 6,2
ATTRIBUTES(GREEN)
    DISPLAY r_clients.id_client USING "###",
            " ",r_clients.first_name,r_clients.last_name," ",
            r_clients.start_date, " ",r_clients.end_date AT 7,2
END IF

# The prepared DELETE operator is used to delete all records with

```



```
# id_client >= 10 (this means the records just added and updated
LET i = 11
LET stmt = "DELETE FROM clients WHERE id_client >= ",i USING "<<<&"
PREPARE p_stmt_10 FROM stmt
EXECUTE p_stmt_10
INITIALIZE r_clients.* TO NULL

# Trying to select and display a deleted row with id_client = 11
SELECT c.* INTO r_clients.* FROM clients c WHERE c.id_client = i
IF status = NOTFOUND -- this option should be triggered,
-- if the row was deleted successfully
THEN DISPLAY "Record is not found because it was deleted" AT 8,2
      ATTRIBUTES (GREEN)
ELSE # if the deleting failed, the row will be selected and status will be 0
      DISPLAY "Deleting failed, the row still exists" AT 8,2 ATTRIBUTES (RED)
END IF

END FUNCTION
```



Constructing a Query by Example

4GL provides the user with an ability to query a database with the help of the user-supplied search criteria. These criteria are entered by the user into the form fields. Such queries are called *queries by example*.

When the program performs a query by example, it converts the user specified values into Boolean expressions that define the search criteria which can be used within the WHERE clause of the SELECT statement.

To declare a query by example, the user must use the CONSTRUCT statement. The syntax of the statement is as follows:

```
CONSTRUCT variable_clause [ATTRIBUTE clause] [Help number]  
[Control Block]
```

All these elements, their usage and variations are described in this chapter.

Since the CONSTRUCT statement is closely connected to the input performed into form fields, it can also be used to restrict the values that can be entered by the user as well as to control the input environment. You should make several steps before you can use the CONSTRUCT statement:

- Define the fields which are linked to the database columns in the form specification file.
- Specify a variable of the character data type in the declaration clause. This variable will get the Boolean expression resulting from the CONSTRUCT statement execution.
- Open and display the screen form by means of OPEN WINDOW...WITH FORM or OPEN FORM and DISPLAY FORM statements.
- Use the CONSTRUCT statement to create a Boolean expression based on the user-entered criteria and to store it to the specified character variable.

The CONSTRUCT statement, when run, activates the form which is the most recently displayed or is displayed in the current window. You can specify which window will be the current one by means of the CURRENT WINDOW statement thus the form in this window will also become the current one. When the program completes the execution of the CONSTRUCT statement, it clears and deactivates the current form.

The process of the CONSTRUCT statement execution can be divided into the following steps:

- The program clears all the screen fields that are specified in the CONSTRUCT field list.
- The program executes the statements that comprise the BEFORE CONSTRUCT control block, if any.
- The screen cursor is positioned to the screen record listed the first in the field list.
- The program waits for the user to input search criteria to the screen fields. If the user leaves a screen field empty, any value in the table column linked to this field will satisfy the search criteria.
- The program executes the statements that comprise the AFTER CONSTRUCT control block, if any.

When the user enters the criteria and presses the ACCEPT key (ESCAPE by default), the CONSTRUCT statement combines all the entered values, forms search criteria represented by a Boolean expression, and stores it to the character variable.



The Boolean expression that results from the CONSTRUCT statement includes additional operators, so you should declare the length of the character variable as several times longer than the total length of all the table columns names used for the query in order to prevent the possible overflow.

The character variable gets the TRUE expression if the user enters no criteria, and all the data in specified tables and columns will be selected when the result of the CONSTRUCT statement execution is applied in the query.

The CONSTRUCT Variable Clause

The *variable clause* of the CONSTRUCT statement is the only obligatory clause and is used to declare the variable to which the search criteria will be stored. It also specifies the columns which will be used for the query.

The syntax of the CONSTRUCT variable clause is given below:

```
CONSTRUCT variable ON columns_list FROM field_list
```

The *variable* stands for the identifier of a variable of CHAR or VARCHAR data type which will store the Boolean value.

The column list can contain one or more column names separated by commas. If the column name is not unique, you should prefix it with the table name. You can also add the owner name and the database name. If you want to create a query on all the columns of a table, you can use the *table.** structure in the column list.

The FROM sub-clause specifies the form fields which should receive the search criteria from the user. It can include one or more fields separated by commas. The field names can be prefixed by the table prefix (the name of the tables to which they are linked). A field can also be preceded by the screen array prefix which should contain the number of the screen array row, e.g.: *scr_arr[5].f001*. You can also use the asterisk to specify all the fields of a record and THRU keyword to specify a subset of fields here.

The CONSTRUCT *variable list* creates temporary bounds between the specified screen fields and database columns, thus identifying the database columns to which the search criteria will be applied. It is important, that the fields and the corresponding database columns should be of the same or compatible data types. The order in which you list the fields in the FROM clause specifies the order in which the cursor moves through the form fields. If you specify a screen array in the field list, you can refer to only one screen record and have to specify its row number.

An example of the simplest CONSTRUCT statement application can look as follows:

```
CONSTRUCT quer ON col_1, col_2 FROM field_1, field_2
DISPLAY "Your criteria are: ", quer
```

The DISPLAY statement will show the user how the program has transformed the entered values into a Boolean expression. If the user enters "Name" to the column named *col_1* and "20" to the column *col_2*, the variable *quer* will get the value "*col_1='Name' AND col_2=20*".

The ON Keyword and the Columns List

The ON keyword is used to specify a list of database columns for which the user-entered search criteria will be applied. It is not necessary that the columns belong to the same table; they can belong to different ones.



The referenced table can be located in the current database, in the specified database, or specified in DBPATH environment variable.

You should be attentive when specifying the ON and the FROM clauses. The number of elements which are explicitly or implicitly (screen record members and screen array elements) specified in the FROM clause must match the number of columns listed in the ON clause.

You can use the *table.** notation if you want to include all the columns of the table into the query without changing the order of reference to them. If you have changed the order of the table columns by means of the ALTER TABLE statement, don't forget to make all the necessary changes in queries and screen forms, otherwise an error may occur.

If the FROM clause includes the names of screen array elements, you must specify the row in which the construct will be activated. To do this, use the construction *screen_record[/line].field_name*:

```
CONSTRUCT my_querr ON cust.* FROM custom[4].*
```

The */line* must be represented by a positive non-zero integer.

The BY NAME Clause

The variable clause can have different syntax, if the names of the columns and the names of the fields are the same. The syntax in this case is as follows:

```
CONSTRUCT BY NAME variable ON columns_list
```

If you use the BY NAME clause, you have to list only the columns you want to query and you don't have to specify the field names, e.g.:

```
CONSTRUCT BY NAME my_querr ON id, f_name, l_name, age
```

The user will be allowed to enter search criteria in the fields with names *id*, *f_name*, *l_name*, *age*, and the criteria will be applied to the database columns which have the same names. If the field names and the column names do not match, an error will occur.

The example given above is synonymous to the following structure:

```
CONSTRUCT my_querr ON id, f_name, l_name, age FROM id, f_name, l_name, age
```

If the BY NAME clause is used to associate column names with fields of a screen array, the first row of the screen array will be used by the CONSTRUCT statement. If you want the statement to select another row, you have to use the FROM clause instead of the BY NAME clause:

```
CONSTRUCT my_querr  
ON id, f_name, l_name, age FROM scr_arr[3].*
```

This statement will make the program activate the third row of the screen array named *scr_arr* for input.



	<p>Note: When you use the BY NAME clause, you cannot add an owner name, a pathname, or a database server name as prefixes. If the alias specification needs these elements, you must use the FROM clause.</p>
---	--

The ATTRIBUTE Clause

The general principle and purposes of the attribute clause were discussed in previous chapters. The ATTRIBUTE clause used within the CONSTRUCT statement applies display attributes to the values that are entered by the user to the form fields. It covers all the form fields listed in the FROM clause or specified implicitly in the BY NAME clause, but the attributes are in effect only during the current activation of the form. The form gets its previous attributes as soon as the CONSTRUCT statement is executed and the form becomes deactivated.

When the ATTRIBUTE clause is present in the CONSTRUCT statement, it overrides the default form fields attributes listed in the form specification file. The NOENTRY and AUTONEXT attributes, specified in the form file, are also ignored.

If the AUTONEXT attribute is specified for a form field, and the user enters a criterion during the CONSTRUCT statement execution, the keys may pass the field delimiter, but the cursor won't move to the next field. The AUTONEXT attribute is ignored in order to give the user the ability to query for large ranges, specify wide alternatives, etc.

The CONSTRUCT attributes temporary override any display attributes, specified within the OPEN WINDOW, OPTIONS, or INPUT statements.

In the example given below, the values entered by the user into the screen fields are displayed in green colour and reversed graphics:

```
CONSTRUCT BY NAME my_querr ON personal.*  
ATTRIBUTES (REVERSE, GREEN)
```

The HELP Clause

The HELP clause is used to specify the number of the help message which will be displayed when the user presses the Help key during the input of the query criteria. The message will be displayed in the Help window. The default Help key is CONTROL-W, but it can be changed by means of the OPTIONS statement.

The help file should be created as it was described in the [HELP](#) clause section of the chapter "User interaction statements" and specified in the HELP FILE clause of the OPTIONS statement.

If you want to create field-level help messages, use the ON KEY clause of the input control block and the [SHOWHELP\(\)](#) function instead.

```
CONSTRUCT my_querr ON custom.* FROM personal.* HELP 105
```

The CONSTRUCT Input Control Blocks

The CONSTRUCT statement may include one or several control blocks that specify the program action during, before, or after the CONSTRUCT statement execution. Each CONSTRUCT control block must include at least



one statement and an activation clause specifying when this statement is to be executed. This block is similar to the control block of the INPUT statement, as the CONSTRUCT statement also uses form fields for input.

They are as follows:

- The BEFORE CONSTRUCT and AFTER CONSTRUCT blocks specify the actions that are to be performed before the user is permitted to enter the search criteria or after he enters them and presses the Accept key;
- The BEFORE FIELD and AFTER FIELD blocks specify the actions that are to be performed before or after a criteria is entered to a specified form field.
- The ON KEY block is used to define which actions are to be performed when the user presses one of the specified keys.
- The ON ACTION clause used to define which actions are to be performed when the user presses a button associated with an action (e.g., CONFIG = "action {Button name}").

The control blocks may include the CONTINUE CONSTRUCT, EXIT CONSTRUCT, NEXT FIELD clauses, and almost all of the 4GL and SQL statements.

The program executes the program blocks in the next consequence:

- BEFORE CONSTRUCT
- BEFORE FIELD
- ON KEY
- ON ACTION
- AFTER FIELD
- AFTER CONSTRUCT

If the CONSTRUCT statement includes no control blocks, the program waits for the user to fill the fields with values and does not perform any additional actions. The CONSTRUCT statement terminates as soon as the user finishes entering the values and presses the Accept key, and the program control passes to the next statement.

When you include at least one of the CONSTRUCT input control blocks into your CONSTRUCT statement, you have to put the END CONSTRUCT keywords after the last statement of the last control block. These keywords separate the statements of the control block from the statements that are to be executed after the CONSTRUCT statement.

The BEFORE CONSTRUCT Clause

The BEFORE CONSTRUCT block is used to specify a set of actions that the program must perform before the user starts entering criteria into the screen fields. The statements specified in the BEFORE CONSTRUCT block are executed only once, and the CONSTRUCT statement can include only one such block.

When the program starts executing the CONSTRUCT statement, it clears all the fields that are listed in the fields list, but the BEFORE CONSTRUCT block can be used to insert default values into them. To do this, you can use the DISPLAY TO statement. For example, the following BEFORE CONSTRUCT block displays values stored of the program record *mem* and variable *id* to the screen record *members*:

```
CONSTRUCT BY NAME const_v ON customer.*  
BEFORE CONSTRUCT  
    DISPLAY mem.* , id TO members.*  
...
```



```
END CONSTRUCT
```

You can use the NEXT FIELD clause to specify the form field the criteria input should be started from:

```
CONSTRUCT BY NAME const_v ON customer.*  
BEFORE CONSTRUCT  
    DISPLAY us_name TO user_n  
    NEXT FIELD age  
    ...  
END CONSTRUCT
```

The BEFORE FIELD Clause

The BEFORE FIELD block is used to specify a set of actions that the program must perform before the cursor moves to a specified screen field. The BEFORE FIELD block is executed any time the cursor is positioned to the specified field before the user is allowed to input values. The CONSTRUCT statement can include several BEFORE FIELD blocks, and one field can have more than one of them. One BEFORE FIELD clause can specify several fields to which you want to apply identical actions, the field names should be separated by commas.

The BEFORE FIELD block can be used to give the user the comments on what information they are to enter, or used together with the NEXT FIELD clause in order to restrict the access to the field:

```
CONSTRUCT BY NAME const_v ON customer.*  
BEFORE FIELD passport  
    IF us_name NOT MATCHES "Admin"  
        THEN NEXT FIELD name  
    END IF  
    MESSAGE "Enter the passport number"  
    ...  
END CONSTRUCT
```

The ON KEY Clause

As with the other statements like PROMPT or INPUT, you can use one or several ON KEY blocks to indicate the actions that the program must perform, if the user presses one of the specified keys.

The restrictions and the rules that you have to keep to when specifying the activation keys of the ON KEY block are the similar to those described in the [ON KEY](#) section of the Chapter "User interaction statements".

The following example illustrates how a help message can be called and displayed by means of the ON KEY block:

```
CONSTRUCT que ON custom.* FROM arr_pers[5].*  
Attribute (BOLD, GREEN)  
  
BEFORE CONSTRUCT  
    DISPLAY "Press F1 for Help, CONTROL-E to skip the input"  
  
    ON KEY (F1)  
        CALL showhelp(105)  
    ON KEY (CONTROL-E)  
        EXIT CONSTRUCT
```



```
END CONSTRUCT
```

When the user activates one of the ON KEY blocks during the search criteria entry, the 4GL performs the following operations:

- Suspends the current input;
- Saves the values that the user has already input to the buffer;
- Executes the statements specified in the activated ON KEY clause;
- Restores the values saved to the input buffer to the screen field;
- Resumes the input and positions the cursor to the field where it was just before the user activated the ON KEY block, and puts the cursor to the end of the entered value.

You can influence the process of the ON KEY block execution. For example, you can include the NEXT FIELD keywords to change the field to where the input will be restored, or change the value saved to the buffer.

The ON ACTION Clause

The ON ACTION clause is treated very much like the ON KEY clause; it activates the statements that are to be executed when the user presses a button or another widget (e.g., radio button, check box, hotlink, etc.) associated with the corresponding action, whereas the ON KEY clause does not necessarily refer to a form widget and can be activated with the help of the keyboard.

The syntax of the ON ACTION clause is as follows:

```
ON ACTION ("action1"[, "action2", "action3", "action4"])
```

```
STATEMENTS
```

The maximum number of actions that can be included to the action list is four.

The *action1...* in this scheme stands for the name of the action specified for the form widget.

As with the ON KEY clause, the ON ACTION clause can contain any SQL or 4GL statements. These statements are executed when the widget within a screen form to which the action is assigned is pressed. When the program starts to process the ON ACTION clause, it deactivates the form used by the CONSTRUCT statement and stores the values entered to this form. When all the statements of the ON ACTION clause are executed, the program control passes to the AFTER CONSTRUCT clause, if any, or to the END CONSTRUCT keywords. No AFTER FIELD clauses are executed after the ON ACTION clause is processed.

If there is an EXIT CONSTRUCT statement in an ON ACTION clause, and this statement is executed, the CONSTRUCT statement is terminated and any AFTER CONSTRUCT clause is ignored.

In the following example, the user can press the button with the "Exit" action, and call a message box asking them if they want to select all the rows of the table:

```
LABEL retry:
```

```
CONSTRUCT con_var ON clients.country, clients.gender FROM country, gender  
ON ACTION ("exit")
```



```
LET ex = fgl_winbutton("Exit", "Do you really want to select all the rows?",  
"No", "Yes|No", "question", 1)  
  
IF ex = "YES" THEN EXIT CONSTRUCT  
  
ELSE GOTO retry  
  
END IF  
  
END CONSTRUCT
```

The AFTER FIELD Clause

AN AFTER FIELD block is executed when the cursor moves from the specified field. The cursor leaves the field when the user presses any arrow key, a RETURN key, an Accept key, or a TAB key. If the user presses RIGHT or LEFT arrow keys, the cursor will first move left or right through the value that has already been input and only then moves to another field. One AFTER FIELD clause can specify several fields to which you want to apply identical actions, the field names should be separated by commas.

The following program checks the value entered to the field *cus_age* and makes the user re-enter it if the customer turns out to be too young:

```
CONSTRUCT que ON custom.* FROM arr_pers[5].*  
Attribute (BOLD, GREEN)  
AFTER FIELD age  
LET pers.age=GET_FLDBUF(age)  
IF pers.age<15 then  
    MESSAGE "There are no customers who are younger than 15"  
CONTINUE CONSTRUCT  
END IF  
END CONSTRUCT
```

You cannot specify more than one AFTER FIELD blocks for the same screen field.

The AFTER CONSTRUCT Clause

You can use the AFTER CONSTRUCT clause to specify the statements that the program will perform after the user enters the last search criteria and presses the Accept key, but before the 4GL creates a string with a Boolean expression.

The AFTER CONSTRUCT block may prove useful, if you want to check, save, or validate the values entered by the user.

This block may contain the NEXT FIELD or CONTINUE CONSTRUCT blocks if it turns out that the user has to re-enter some of the criteria. However, you should be attentive when including these statements to the AFTER CONSTRUCT block. It is advisable, that they are located within a conditional statement, otherwise, the user won't be able to finish the input and quit the form.

The following snippet of a source code checks if the user has entered any value to the screen form. If the user hasn't entered any criteria, the cursor returns to the form:

```
CONSTRUCT us_querry ON custom.* FROM order_det.*
```



```
AFTER CONSTRUCT
IF NOT FIELD_TOUCHED(order_det.*) THEN
    MESSAGE "You must enter at least one search criteria"
CONTINUE CONSTRUCT
END IF
END CONSTRUCT
```

The CONSTRUCT statement can contain only one AFTER CONSTRUCT clause, which is activated when the user presses the Accept, the Interrupt, or the Quit key (in case the DEFER statement has been executed before).

The program does not execute the AFTER CONSTRUCT block if the DEFER QUIT or DEFER INTERRUPT statements have **not** been executed before the user presses the Quit or Interrupt key, respectively. It is also not executed if the CONSTRUCT statement is terminated by the EXIT CONSTRUCT statement.

	Note: When the result of a conditional clause used in the AFTER CONSTRUCT control block requires the further action of the user, it is more convenient to use the NEXT FIELD command rather than the CONTINUE CONSTRUCT statement.
--	---

Using Built-In Functions and Operators in CONSTRUCT Input Control Blocks.

4gl supports using different built-in functions and operators within the CONSTRUCT statement. All of them have already been described in details earlier, and below there is a table with their short descriptions and links to the corresponding parts of this manual.

Function	Description
<u>FIELD_TOUCHED()</u>	Returns the TRUE value if the user has changed the value that was previously displayed to the specified field. The field isn't treated as touched if the cursor has just moved through the field.
<u>GET_FLDBUF()</u>	Returns character values entered to one or more fields of the active form.
<u>FGL_GETKEY()</u>	Returns an INTEGER value corresponding to the raw value of the key pressed by the user.
<u>FGL_LASTKEY()</u>	Returns an INTEGER value which indicates the raw number of the key that the user pressed the last during the input.
<u>INFIELD()</u>	Returns the TRUE value if the current field name and the name of the field specified in the brackets match

Each form field has one field buffer, which cannot be referenced by two statements simultaneously. If you want to use the same form for several times, it is advisable that you open this form in different windows and under different names, because 4GL creates a separate set of field buffers for each copy of the form.

The CONTINUE CONSTRUCT Statement

The CONTINUE CONSTRUCT statement is very similar to other CONTINUE statements. It can be used in any control clause when you want the program to exit from it and to return the cursor to the screen form.



When the program executes the CONTINUE CONSTRUCT statement, it skips all the statements that follow it in the current control clause, and returns the cursor to the last screen field it was positioned in.

The EXIT CONSTRUCT Statement

The EXIT CONSTRUCT statement is used to skip all the following statements that the CONSTRUCT statement includes and to terminate its execution.

When the program executes the EXIT CONSTRUCT statement, it doesn't ignore the values that have already been input by the user, and constructs a Boolean expression using these values.

When the CONSTRUCT statement is terminated, the program resumes its execution at the statement which follows the END CONSTRUCT keywords.

The NEXT FIELD Clause

When the CONSTRUCT statement is being executed, the screen cursor, by default, moves from field to field in the order determined by the order in of the screen fields listed in the FROM clause of the CONSTRUCT statement.

However, you can influence the order by means of the NEXT FIELD clause. The NEXT FIELD clause is similar to the one of the [INPUT](#) statement and moves the cursor to the field specified after the NEXT FIELD keywords.

Besides the actual field names, you can specify the NEXT FIELD as PREVIOUS or NEXT. In this case, the cursor will move to the field that was specified in the *fields list* before or after the current one, respectively.

The NEXT FIELD clause is necessary mainly if you want the program to deviate from the default field order. When the program encounters the NEXT FIELD clause, it positions the cursor to the specified field immediately and skips all the statements that follow it in its statement block.

You shouldn't use the NEXT FIELD clause to move the cursor to every field of a form. The order of the cursor movement can be regulated by the order of the field names in the *fields list*. However, when you swap the field names in the fields list, don't forget to swap the corresponding column names, also. Otherwise, compile- and runtime errors can occur.

User-Entered Search Criteria.

The CONSTRUCT statement is designed to give the user an opportunity to specify search criteria that the program will apply when retrieving rows from the database.

4GL supports a set of symbols that the user can add to the search criteria in order to make them more accurate. These symbols are relational operators and some of the symbols used by the MATCHES operator (for character values only). Their meanings within the user-entered values are described in the table below:

Symbol	Meaning	Data Type	Examples of an input value
= or ==	Equal to	All simple data types	=345, = TRUE, =
>	Greater than. For character fields, "greater" or "less" then mean <i>later</i> or <i>earlier</i> in the	All simple data types	>x



ASCII sequence, respectively. For DATE and DATETIME values "greater" means <i>later in time</i> , and "less" means <i>earlier in time</i>			
<	Less than	All simple data types	<z
>=	Greater than or equal to	All simple data types	>=a
<=	Less than or equal to	All simple data types	<=b
<> or !=	Not equal to	All simple data types	!=id, <>id
: or ..	Indicates a range. The ".." sign should be used in DATE, DATETIME, and INTERVAL values, because the ":" symbol will be treated as a time-unit separator.	All simple data types	A..z, -10:10
*	Wildcard for any string	CHAR, VARCHAR	*a, a*, *a*, a*b
?	Wildcard for a single character	CHAR, VARCHAR	fan?, ap??, a??e
	Logical OR symbol	All simple data types	x y
[]	List of values	CHAR, VARCHAR	

If the search criteria entered by the user exceed the size of the field, the value is displayed and input to the Comment line. If the comment line contains any string displayed before that, the string is erased.

4GL treats any character that follows the operators listed above as a literal.

To see more detailed information about [relational operators](#) refer to the corresponding chapter; the operators for comparing character values are described in the [MATCHES and LIKE](#) section of the Chapter concerning the conditional statements and operations

Editing During the Search Criteria Input

4GL supports a set of reserved keys which can be used to simplify the edition of user-entered criteria during the input. These keys are:

Key	Effect
CONTROL-A	Switches from insert to type-over mode and back
CONTROL-D	Deletes characters from the cursor position till the end of the current field
CONTROL-H	Moves the cursor one space to the left (equivalent to the LEFT arrow key)
CONTROL-L	Moves the cursor one space to the right (equivalent to the RIGHT arrow key)
CONTROL-R	Redisplays the screen
CONTROL-X	Deletes the characters beneath the cursor

Applying Queries by Example

As we have discussed before, the CONSTRUCT statement is only a tool for creating queries by examples, i.e., queries with user-defined criteria which then are specified as the criteria in the WHERE clause of a SELECT statement.

The CONSTRUCT statement results in a Boolean value stored in a character data type variable. To apply the criteria specified in this variable, you have to make the following steps:



1. Declare a variable to where the constructed criteria will be stored. Also declare a variable that will be used in the PREPARE statement as a statement text variable.
2. Open and display the form in which the CONSTRUCT statement will be executed.
3. Create a CONSTRUCT statement to let the user enter the search criteria.
4. Use the PREPARE statement to prepare a SELECT statement which includes the search criteria supplied by the CONSTRUCT statement. The prepared text must consist of a text of the SELECT statement prepared and of the constructed variable which should follow the WHERE keyword of the prepared SELECT statement. The text of the SELECT statement itself should end after the WHERE keyword and it should not contain query criteria. The statement text and the variable must be separated with a comma.
5. Declare a cursor for the prepared statement.
6. Process the cursor to perform the query and return the retrieved rows.

The following source code text will become the result of these operations performance:

```
{1} DEFINE con_var, prep, sel VARCHAR(200)

{2} OPEN WINDOW myw AT 1,1 WITH FORM "construct"

{3} CONSTRUCT con_var ON custom.* FROM const.*
CLOSE WINDOW myw

{4} PREPARE sel FROM "SELECT * FROM custom WHERE ",con_var

{5} DECLARE cur CURSOR FOR sel

{6} FOREACH cur INTO pers. *
    DISPLAY pers. *
    SLEEP 1
END FOREACH
```

Example

This application allows the user to compose a query and then use it to retrieve rows from a database table.

For this application to work you need to be connected to the database specified in the DATABASE statement. Your database must also contain the tables which can be created by [Databases in 4GL](#) example and populated by [Populating Database Tables](#) example before you can run this application.

```
#####
# Using the CONSTRUCT statement for composing a query
#####
DATABASE db_examples

MAIN
DEFINE
    r_clients  RECORD LIKE clients.*,
    sel_stmt   VARCHAR(1040), -- the variable which will store the text
                           -- of the complete query
```



```

-- it needs to be large to incorporate all
-- the possible criteria
-- the variable used to store the query
-- criteria entered by the user

var_query      CHAR(560),
var_char       CHAR(50),
info          STRING,
flag_exit,
i_row,
i,j,k,l,
num_key       INTEGER,
first_time    BOOLEAN

LET first_time = TRUE -- this indicates, that the CONSTRUCT statement
                      -- specified later is executed for the first time
OPTIONS
    ACCEPT KEY F3,
    INPUT WRAP

IF fgl_fglgui() = FALSE
THEN
    CALL fgl_winmessage("Character mode",
    "You tried to launch the program in character mode." ||
    "\nSome form widgets cannot be displayed and used correctly." ||
    "\nThe program will be terminated." ||
    "\n\nPlease, run the program in GUI mode", "error")
    EXIT PROGRAM
END IF

# We include the CONSTRUCT and all the other statements
# which process the constructed data into the WHILE statement
# so that after creating one query the user would be able
# a number of queries until the Exit key is pressed.
WHILE TRUE

    # We open the form to be used for entering the query criteria
    OPEN WINDOW w_qry_clients AT 1,1 WITH FORM "qry_clients"
        ATTRIBUTE (FORM LINE 2)
    DISPLAY "!" TO hlp
    DISPLAY "!" TO accpt
    DISPLAY "!" TO qit
    DISPLAY "Using the CONSTRUCT statement for specifying the query criteria"
    TO headr ATTRIBUTE(BLUE, BOLD)

    LET flag_exit = FALSE

    # These commented lines are the example of the CONSTRUCT statement
    # which uses only some of the form fields for constructing a query
    # Remove the comment marks and comment the following CONSTRUCT statement
    # to see how it works
{
    CONSTRUCT BY NAME var_query ON
        id_client,
        first_name,

```



```
        last_name,
        gender,
        address,
        id_country,
        start_date,
        end_date
    }

# This CONSTRUCT statement uses all the fields of the form to get the
# query criteria
CONSTRUCT var_query ON
    id_client,
    first_name,
    last_name,
    date_birth,
    gender,
    id_passport,
    date_issue,
    date_expire,
    address,
    city,
    zip_code,
    id_country,
    add_data,
    start_date,
    end_date
FROM s_clients.id_client,
       s_clients.first_name,
       s_clients.last_name,
       s_clients.date_birth,
       s_clients.gender,
       s_clients.id_passport,
       s_clients.date_issue,
       s_clients.date_expire,
       s_clients.address,
       s_clients.city,
       s_clients.zip_code,
       s_clients.id_country,
       s_clients.add_data,
       s_clients.start_date,
       s_clients.end_date

HELP 1
BEFORE CONSTRUCT -- we use this clause to inform the user
                    -- what wildcard symbols they can use
                    -- when specifying the search criteria
IF first_time = TRUE THEN -- this IF loop is used to indicate whether
                            -- the criteria input form is opened
                            -- for the first time
                            -- If it is, a window with instructions will
                            -- appear. If not, the BEFORE CONSTRUCT
                            -- block execution will be terminated
    LET first_time = FALSE
    CALL info()
END IF
```



```
ON KEY (F1)
    CALL info() -- this function gives hints about constructing a query

ON ACTION (qit)
    LET flag_exit = TRUE
    EXIT CONSTRUCT -- this is the only way to exit the program

END CONSTRUCT

CLOSE WINDOW wqry_clients

IF flag_exit = TRUE -- this is true, if the CONSTRUCT is terminated
THEN CLEAR SCREEN -- by pressing the "Quit" button
    EXIT PROGRAM -- so the program may terminate also
END IF

OPEN WINDOW search_crit AT 2,10 WITH FORM "tip_form"
ATTRIBUTE (BORDER)
# After user entered the criteria for search, we need to process them:
# to include them into a SELECT statement - into the WHERE clause
# and then run the query
DISPLAY "!" TO ok
DISPLAY "Using the CONSTRUCT statement to specify the query criteria"
    AT 5,2 ATTRIBUTE(BLUE, BOLD)
DISPLAY "Variable sel_stmt contains the complete text of the query"
    AT 6,2 ATTRIBUTE(RED, BOLD)

LET sel_stmt = "SELECT * FROM clients WHERE ",var_query CLIPPED, " ",
    "ORDER BY id_client"

# We display the value of the sel_stmt variable to a multisegment field
# If the value is too long to be displayed to one row, it will be
# displayed to several rows

DISPLAY sel_stmt TO disp_field

LABEL press_ok2:
    LET num_key = fgl_getkey()
    IF num_key = 343      -- 343 corresponds to the Accept key,
                          -- specified for the OK button in the
                          -- tip_form form file
    THEN CLOSE WINDOW search_crit

    ELSE GOTO press_ok2
    END IF
    CLEAR SCREEN

WHENEVER ERROR CONTINUE

# Then we use the SELECT statement with the criteria gained from the
# CONSTRUCT statement to prepare a query
PREPARE p_sel_stmt FROM sel_stmt
LET i = status
```



```
WHENEVER ERROR STOP

# If the statements cannot be prepared due to an error
# we display the corresponding error message and number
IF i <> 0
THEN DISPLAY "Error of search criteria entry: status = ",i USING "-<<<&"
      AT 23,1 ATTRIBUTE(RED, BOLD)
DISPLAY "Press any key to continue . . ." AT 24,1
LET num_key = fgl_getkey()
CLEAR SCREEN
CONTINUE WHILE

END IF

# If the PREPARE statement was successful,
# we use the prepared statement in the cursor declaration
DECLARE c_sel_stmt CURSOR FOR p_sel_stmt

# We use the cursor to select and display the values
# which correspond to the specified search criteria
LET i_row = 2
FOREACH c_sel_stmt INTO r_clients.*
      DISPLAY r_clients.id_client USING "###", " "
            ,r_clients.first_name,r_clients.last_name,r_clients.date_birth,
            " ",r_clients.gender
      DISPLAY r_clients.id_passport, " ",r_clients.date_issue,
            " ",r_clients.date_expire," ",r_clients.address CLIPPED
      DISPLAY r_clients.city, " ",r_clients.zip_code, " ",r_clients.add_data
            CLIPPED," ",r_clients.start_date, " ",r_clients.end_date
      DISPLAY ""
      SLEEP 1
      LET i_row = i_row +1
END FOREACH

# If the query didn't retrieve at least one row
# which is possible, if the user entered invalid search criteria
# we display the corresponding message and return to the CONSTRUCT
# statement
IF i_row = 2
THEN
CALL fgl_winmessage("Attention",
"Data not found", "Error")

      ELSE CLEAR SCREEN
END IF

END WHILE -- this is the end of the loop in which allows the user
-- to execute the CONSTRUCT statement any number of times

END MAIN

#####
```



```

# This function shows the user the symbols allowed while creating a query
#####
FUNCTION info()
DEFINE
    info      STRING,
    num_key   INT

OPEN WINDOW w_mess AT 2,10 WITH FORM "tip_form"
    ATTRIBUTE (BORDER)

LET info = "(=) Equal to e.g. ID [=10      ],
            "(<) Not equal to e.g. ID [<>5      ],
            "(<) Less than e.g. ID [<10      ],
            "(>) Greater than e.g. Birth Date [>01/01/1970      ],
            "(<=) Not greater than e.g. Birth Date [<=01/01/1970      ],
            "(>=) Not less than e.g. Registered [>=01/01/1999      ]
            "(:) Range e.g. ID [1:5      ],
            "(|) Logical OR e.g. ID [3|6      ),
            "(*) Wildcard for any string e.g. Last Name [S*      ],
            "(?) Single-character wildcard e.g. First Name [Ya??      ],

DISPLAY "Enter the data search criteria. You can use " AT 6,4
    ATTRIBUTE(BLUE, BOLD)
DISPLAY "these symbols to search for data values: " AT 7,4
    ATTRIBUTE(BLUE, BOLD)

DISPLAY info TO disp_field
DISPLAY "!" TO ok

LABEL press_ok3:
LET num_key = fgl_getkey()
    IF num_key = 343          -- 343 corresponds to the Accept key,
                                -- specified for the OK button in the
                                -- tip_form form file
        THEN CLOSE WINDOW w_mess
        ELSE GOTO press_ok3
    END IF
END FUNCTION

```

Form Files

This is the form file 'tip_form' :

```

DATABASE formonly

SCREEN SIZE 25 BY 80

{
\gp-----q\g
\g|     | \g
\g|     | \g
\g|     | \g
\g|     | \g

```



This is the form file 'qry_clients' :

```
DATABASE db_examples
SCREEN SIZE 27 BY 80

-----[102]-----]
ID:\g[f00      ]
First Name:\g[f01      ]
Last Name:\g[f02      ]
Birth Date:\g[f03      ]
Gender:\g[f04      ]
-----[103]-----]
Passport:\g[f05      ]\gIssued on:\g[f06      ]\gExpires on:\g[f07      ]
-----[104]-----]
ss:\g[f08      ]
City:\g[f09      ]
Zip:\g[f10      ]
Country:\g[f11  ][f16]
-----[105]-----]
Registered:\g[f13      ]\g
Dismissed:\g[f14      ]
-----[106]-----]
1      [b02      ]      [b03      ]
-----[107]-----]
d\g
```



{

```
TABLES clients

ATTRIBUTES
f16 = FORMONLY.country_name TYPE CHAR,NOENTRY;

f00 = FORMONLY.id_client,      COMMENTS=" Enter client ID number";
f01 = FORMONLY.first_name,    COMMENTS=" Enter the first name";
f02 = FORMONLY.last_name,     COMMENTS=" Enter the last name";
f03 = FORMONLY.date_birth,   widget = "calendar",
    COMMENTS=" Enter the date birth";
f04 = FORMONLY.gender,      widget = "combo", include = ("male", "female"),
    COMMENTS=" Enter the gender";
f05 = FORMONLY.id_passport,  COMMENTS=" Enter the passport number";
f06 = FORMONLY.date_issue,   widget = "calendar",
    COMMENTS=" Enter the date the passport was issued";
f07 = FORMONLY.date_expire,  widget = "calendar", PICTURE="##.##.####",
    COMMENTS=" Enter the date when passport expires";
f08 = FORMONLY.address,
    COMMENTS=" Enter the street address, apartment, housing, etc.";
f09 = FORMONLY.city,         COMMENTS=" Enter the city";
f10 = FORMONLY.zip_code,    COMMENTS=" Enter the zip code for this address";
f11 = FORMONLY.id_country,  widget = "combo", include = (276, 804,826),
    CLASS = "combo", COMMENTS="Enter the country id";
f12 = FORMONLY.add_data,    WORDWRAP,COMMENTS=" Enter the additional data";
f13 = FORMONLY.start_date,  widget = "calendar", PICTURE="##.##.####",
    COMMENTS=" Enter the date of registered";
f14 = FORMONLY.end_date,    widget = "calendar", PICTURE="##.##.####",
    COMMENTS=" Enter the date of deregistered";

b01 = formonly.hlp,
    config = "F1 {How to Compose a Query}", widget = "button";
b02 = formonly.accpt,
    config = "F3 {Compose a Query}", widget = "button";
b03 = formonly.qit,
    config = "qit {Quit}", widget = "button";

101 = formonly.headr, config = "", COLOR= GREEN BOLD,
    widget = "label", CENTER;
102 = formonly.tabname, config = "CLIENTS DATA", COLOR= REVERSE CYAN,
    widget = "label", CENTER;

INSTRUCTIONS
DELIMITERS "[ ]"
SCREEN RECORD s_clients (id_client THRU end_date)
```



4GL ADVANCED

This section will teach you to use some additional 4GL features which are not necessary for including into every 4GL program, but which may make your programs more user friendly and the programming and maintenance process easier. These are web functions which allow you to use some ready - made functions in your program, reports which facilitate database interactions and error handling which makes your program more stable and easier to maintain.



Web Services

This chapter deals with one of the most important concepts in the IT world these days - Web Services. You will get familiar with the notion of web services, learn about the main reasons for their extensive use in software development and about technologies employed to make their work possible. You will also learn how to develop, compile and deploy your own web services using Querix tools and take advantage of those provided by others. This chapter will cover the most basic notions and activities, for more information about the web service creation, compilation, deployment and usage refer to the "Web Services Getting Started" guide.

The notion of Web Services

One of the major concerns that software engineers have always had is how to create software which is simple to use, update and integrate with other applications. The last problem has become an especially burning issue over the past few years. Various methods have been tried in an attempt to address it, and the latest solution that has been found is the use of web services.

What is a Web Service?

A web service can be regarded as a container for one or several functions, each performing a specific task, which are made public and can be accessed by other applications over a network using open network protocols such as HTTP. These functions are called *web service operations*, and the tasks they perform are known as *methods*. Each method in a web service operation has a number of variables that receive values from calling routines. These variables are called *properties* or *parameters*. Collectively, *properties* and *methods* refer to the interface of a web service.

In Q4GL a web service operation is represented by a 4GL function slightly different in syntax, if compared to a normal 4GL function. Thus a web service in Q4GL is a set of web functions compiled into a web service. In its turn a web service is similar to a 4GL program, though unlike a program, a web service has no MAIN...END MAIN program block. Its deployment also differs from the deployment of a regular 4GL program. All these peculiarities will be discussed later in this chapter.

The process of applications exchanging web services is based upon the *request-response* pattern. The application which requests a web service is called a *service consumer*, and the one which offers it is known as a *service provider* (it must not be confused with a company that provides the Internet access).

To expose a web service to other applications, that is to make it a service provider, you must deploy it to a web server along with its description in a special language called WSDL (*Web Service Description Language*). Besides, there are catalogues in the Internet where you can publish a link to the server on which your web service resides, so that others could locate and make use of it. You can search the same catalogues for web services you need.

If you want to use a published web service, you must develop a special program called a *web service client*. This program needs a web service meta data description file in the XML (Extensible Markup Language) format with some basic information about the web service, and also a web service client stub - a function used to invoke the required web service.

To understand it better, suppose Company X has created a web service with a function (or operation) returning weather forecast for a particular city when its name is given. It places this service on a web server in the Internet and gives a reference to it in a web services public directory. Company Y finds this link,



creates a web service client application to utilize this service, and sends a request to the web service provider with the name of a city as its parameter. The provider receives the request, executes the function and returns the result to the web service client.

Core Elements Behind a Web Service

The practical application of web services is made possible thanks to a number of technologies listed in the following table and briefly explained after it.

Technology	Full name	Description
XML	Extensible Markup Language	Provides a universal, platform independent format for data transition which makes the applications ignore the internal code of each other so that they can communicate by following the rules they both can understand.
SOAP	Simple Object Access Protocol	An XML based, open, platform independent communication protocol (like HTTP) used for the exchange of messages over the Internet
WSDL	Web Services Description Language	An XML based format used for describing and locating Web Services
UDDI	Universal Description, Discovery and Integration	A directory service for registering and searching for Web Services

The process of information exchange between service agents by means of the technologies described above goes like this.

1. The service provider sends a message in WSDL format to the UDDI directory specifying description and location of the web service.
2. The service consumer issues a query to the same directory in order to find the required web service and learn how to communicate with it.
3. A message in WSDL format is sent to the service consumer with the information on the expected requests and responses.
4. Using the received information, the service consumer forms a WSDL request to the provider.
5. The provider receives the WSDL request from the consumer and sends back a response in the same format.

Throughout the whole process SOAP envelopes the messages and sends them via an open Internet protocol, most likely HTTP.

What You Need to Use Web Services

To be able to use web services Lycia alone is not enough. You will need the following to use them:

- A web server up and running either on the same machine where Lycia is installed or on a machine to which you have access. It can be WebSphere, Tomcat, etc..
- Microsoft Visual Studio installed on the same machine where Lycia is installed, in case if you use Windows, or other C compiler used for other systems. You can find the list of the required C compilers in Lycia documentation.



Creating a Web Service Operation

To create a web service and be able to use it, you must create a web service proper which must contain one or more web functions and the web service client, which will invoke the web functions of the web service.

To create a web service using Lycia Studio you need to go to **File -> New -> New Web Service** main menu option. This will create an object similar to a 4GL program to which you can link 4GL source files containing the web functions. For more information about how to create a web service refer to the "Web Services Getting Started" guide.

An individual web service, as has already been said at the beginning of the chapter, can be essentially regarded as a container for one or several web functions with a definite task. The web functions that can only be accessed by other web functions in the same web service will be private or hidden to the outside world. Those that are exposed to the outside world, i.e. made public or visible to other applications, will become *web service operations* - they can be accessed by the web service clients. Correspondingly, to create a web service it is necessary to declare at least one web function (or web service operation).

An ordinary function declaration the way it is done in 4GL is not sufficient for web services. For example, the following code will not work with a web service:

```
FUNCTION func_name( )
...
END FUNCTION
```

To make this function web service compatible you should add two keywords - WEB and RETURNS.

```
WEB FUNCTION func_name RETURNS (returned_value)
```

The keyword WEB indicates that the function is a web service operation and it can be seen and accessed by other applications. The *func_name* is the name of the web service operation. The keyword RETURNS specifies variables that will receive values upon the operation completion. They must be enclosed within a set of parentheses. The data type of these returned variables must be clearly defined. Here is an example of a web service function declaration:

```
WEB FUNCTION add_value(arg_int) RETURNS(my_int_ret)

DEFINE arg_int INT, my_int_ret INT

...

END FUNCTION
```



Note: The web service extensions for a function does not prevent it from being used as a normal 4GL function in a 4GL program.

The RETURNS Keyword

The RETURNS keyword in a web service operation specification is somewhat different from the [RETURNING](#) keyword in a 4GL function declaration. It applies more restrictions on returned variables.



An ordinary 4GL function can return different number of variables of different data types depending on the condition.

For example, a 4GL function called *f1()* in the following code will return nothing (the first RETURN statement), an integer value of the *my_int1* variable (the second RETURN statement), integer values of *my_int1* and *my_int2* variables (the third RETURN statement). Which of these variables will get a value depends on the value of the *my_condition* variable. The data types, number and order of the returned values for a normal 4GL function are defined by the RETURNING clause of the CALL statement invoking this function and by the RETURN statement within the function.

```
FUNCTION f1()

DEFINE my_condition, my_int1, my_int2 INTEGER

... IF my_condition < 0 THEN

RETURN END IF

... IF my_condition > 1 THEN

RETURN my_int1 END IF

... IF my_condition = 1 THEN

RETURN my_int1, my_int2 END IF
...

END FUNCTION
```

The specification above would be impossible with a web service operation. The programmer should specify the number and data types of the returned values directly in the function specification. They must be explicitly specified within the parentheses following the RETURNS keyword and their data types in a DEFINE statement. At the same time the function can still contain the RETURN statement which actually returns the values.

```
WEB FUNCTION add_value(arg_int) RETURNS(my_int_ret)

DEFINE arg_int INT, my_int_ret INT

LET my_int_ret = 100

LET my_int_ret = my_int_ret + arg_int

RETURN my_int_ret

END FUNCTION
```

The web service operation *add_value* accepts a value passed to *arg_int* argument, processes it and returns a value of *my_int_ret* specified as the return variable.



A web service operation may require no values, one or several values to be returned. If the RETURN statement is not present in the function body, the variable specified after the RETURNS keyword will *implicitly* get a value, as illustrated by a slightly modified code above.

```
WEB FUNCTION add_value(arg_int) RETURNS(my_int_ret)

    DEFINE arg_int INT, my_int_ret INT

    LET my_int_ret = 100
    LET my_int_ret = my_int_ret + arg_int

END FUNCTION
```

The variable *my_int_ret* in the RETURNS clause will receive a value despite the fact that the function code doesn't contain the RETURN statement.

The same result will be received if the RETURN keyword is not followed by any variable names: a value will still be assigned to *my_int_ret* and thus it will still be returned.

```
WEB FUNCTION add_value(arg_int) RETURNS(my_int_ret)

    DEFINE arg_int INT, my_int_ret INT

    LET my_int_ret = 100
    LET my_int_ret = my_int_ret + arg_int

    RETURN

END FUNCTION
```

If the number of returned variables after the RETURNS keyword is greater than the number of variables in the RETURN statement, the program will return only the value or values specified in the RETURN statement. The code below demonstrates this.

```
WEB FUNCTION add_value(arg_int1) RETURNS(my_int_ret1, my_int_ret2)
    DEFINE arg_int1, my_int_ret1, my_int_ret2 INT
    LET my_int_ret1 = arg_int1 + 100
    LET my_int_ret2 = arg_int1 + 50
    RETURN my_int_ret1
END FUNCTION
```

This function will return 110 - the value of the *my_int_ret1* variable in the RETURN statement.

On the other hand, when there are fewer variables in the RETURNS clause than values returned by the function, it will cause a runtime error. The function *add_value* in the code below has no variables in the RETURNS clause, and one returned variable in the RETURN statement. When the function is executed, an error will occur.

```
WEB FUNCTION add_value(arg_int1) RETURNS()
    DEFINE arg_int1, my_int_ret1, my_int_ret2, my_int_ret3 INT
        LET my_int_ret1 = arg_int1 + 100
        LET my_int_ret2 = arg_int1 + 50
```



```
LET my_int_ret3 = arg_int1 + 10
RETURN my_int_ret3
END FUNCTION
```

Note that the modifications in a normal 4GL function declaration to make it web service compatible do not affect ordinary function calls, and you can still call this function from 4GL code just as you invoke a usual 4GL function.

```
...
DEFINE a INT
LET a = 3
CALL add_value (a)
...
```

Besides, one web function can call another if they are located in the same web service:

```
WEB FUNCTION add_value(arg_int) RETURNS(my_int_ret)

...
END FUNCTION

WEB FUNCTION add_value2(arg_int2) RETURNS (my_int_ret2)
...
CALL add_value(arg_int2)
..
END FUNCTION
```

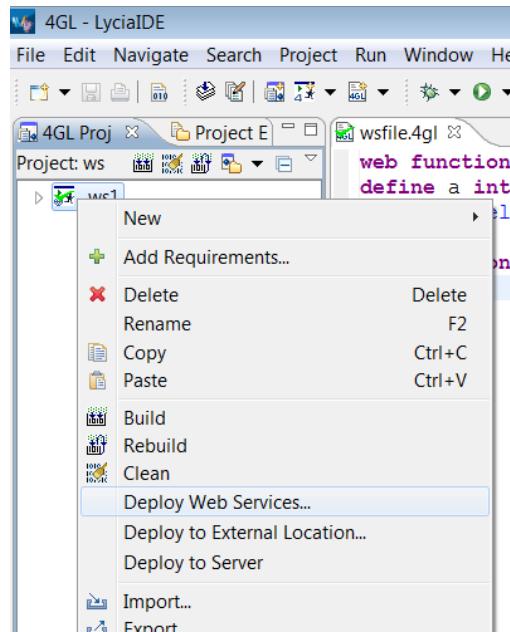


Note: A web service **must not** include a module with the MAIN program block; it can only include a number of modules containing functions.

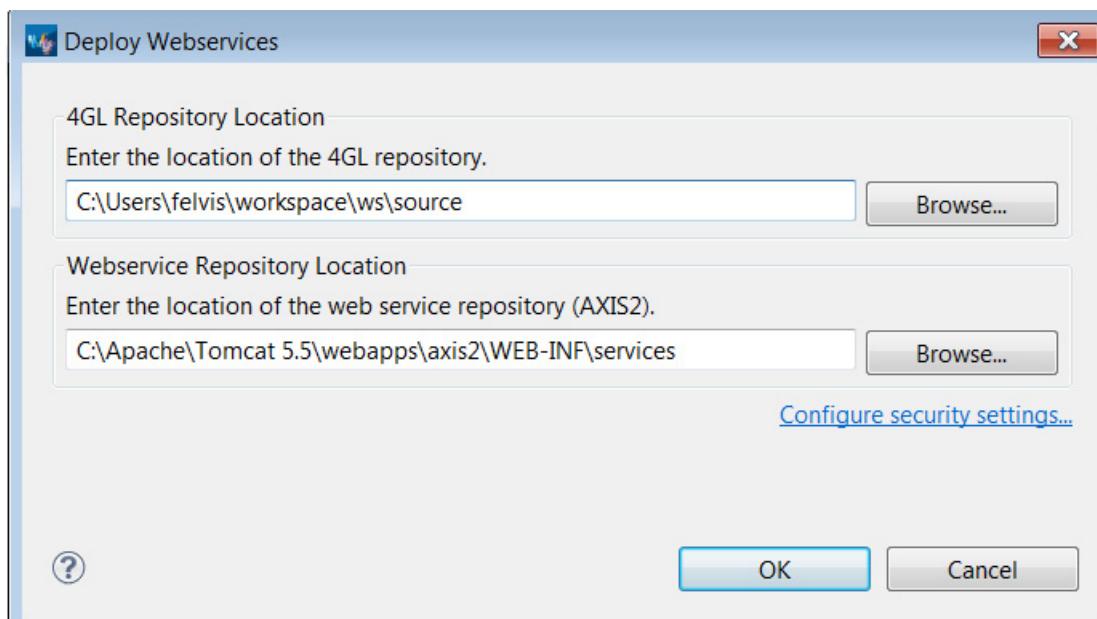
Compiling and Deploying a Web Service

After a web service has been created, it must be compiled and deployed to a web server. Compiling a web service in Lycia Studio is in no way different from compiling a usual 4GL program. To compile a web service using command line tools you need to use the specific options which are outside the scope of this manual. For more information about how to compile a web service both in Lycia Studio and in the command line environment refer to the "Web Services Getting Started" guide.

The deployment of a web service is possible only if you have the access to a web server where tool for web services deployment is installed (i.e. Tomcat software). To deploy a compiled web service in Lycia Studio you should first right-click its name in the 4GL Program view and select the menu option **Deploy Web Services**. This option appears only for a web service object, it is not available for a normal 4GL program.



When it is done, you will be presented with a dialog. Here you should specify the path to the repository where the 4GL file with the web service operation or operations will be stored, and the path to the web service repository on the server. After pressing the OK button, the web service is deployed to the server.



The web service repository location is the location where you have your Tomcat or WebSphere installed. This example is based on the Tomcat installation, the web service should be deployed to ...\\axis2\\WEB-INF\\services folder of your Tomcat installation. For more information about Tomcat and its functionality see the documentation shipped with this software. You can also see the "Web Services Getting Started" guide for the details.



Note: If there is no "axis2" folder in your Tomcat installation, copy "axis2" folder from Lycia home directory and put it into "webapps" folder of your Tomcat. Then deploy your applications to the path described above within this directory.

To verify that the web service is now on the server, you should open a web browser and type in the address line an address in the following format:

`http://<hostname>:<port_number>/<axis2_context_root>/<webservice_repository_folder>`

For example, if you are using a web server on your machine locally (that is if the Tomcat is installed on your local machine as in the example above), this address might look as follows:

`http://localhost:8080/axis2/services`
or
`http://localhost:8080/axis2/services/listServices`

if you use Tomcat 7.* and above. For Tomcat 7.* and above the list of services can be accessed from `http://localhost:8080` by using the graphical interface and selecting first "Manage App" button, then clicking on the "/axis2" application in the list and select "Services".



Note: To access the applications list you may need to specify the user and password for the web server. For the details see Tomcat documentation.

As a result, you will see a web page with the list of all the services available along with the operations they perform. The name of the service you have deployed must be included in that list.

Available services

Version
Service EPR : <http://localhost:8080/axis2/services/Version>

Service Description : Version
Service Status : Active
Available Operations

- getVersion

add_val
Service EPR : http://localhost:8080/axis2/services/add_val

Service Description : add_val
Service Status : Active
Available Operations

If now you click on your web service link, you will be presented with the description of the web service in the WSDL format which is created automatically when the service is being deployed.

```

<?xml version="1.0" encoding="UTF-8"?>
- <wsdl:definitions xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/" xmlns:ns1="http://org.apache.axis2/xsd" xmlns:ns="http://wsoutput.querix.com"
  xmlns:wsaw="http://www.w3.org/2006/05/addressing/wsdl" xmlns:http="http://schemas.xmlsoap.org/wsdl/http" xmlns:xs="http://www.w3.org/2001/XMLSchema"
  xmlns:ns1="http://wsoutput.querix.com" xmlns:ns2="http://schemas.xmlsoap.org/wsdl/mime/" xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/" xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
  ><wsdl:documentation>add_val</wsdl:documentation>
- <wsdl:types>
  - <xsd:schema attributeFormDefault="qualified" elementFormDefault="qualified" targetNamespace="http://wsoutput.querix.com">
    - <xsd:element name="add_value">
      - <xsd:complexType>
        - <xsd:sequence>
          - <xsd:element minOccurs="0" name="args0" type="xsd:int" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
    - <xsd:element name="add_valueResponse">
      - <xsd:complexType>
        - <xsd:sequence>
          - <xsd:element minOccurs="0" name="return" type="xsd:int" />
        </xsd:sequence>
      </xsd:complexType>
    </xsd:element>
  </xsd:schema>
</wsdl:types>
- <wsdl:message name="add_valueRequest">
  <wsdl:part name="parameters" element="ns:add_value" />
</wsdl:message>
- <wsdl:message name="add_valueResponse">
  <wsdl:part name="parameters" element="ns:add_valueResponse" />
</wsdl:message>
- <wsdl:portType name="add_valPortType">
  <wsdl:operation name="add_value">
    <wsdl:input message="ns:add_valueRequest" wsaw:Action="urn:add_value" />
    <wsdl:output message="ns:add_valueResponse" wsaw:Action="urn:add_valueResponse" />
  </wsdl:operation>
</wsdl:portType>
```



Using Tomcat 7.*

If you use Tomcat version 7 and above there are some differences in the interface and

Accessing an External Web Service

To be able to utilize a web service published by a third party or to enable your clients to use your web services, you need to create a special program called a web service client. This program requires an XML meta data definition file in the WSDL format which contains the most important information about the given web service, and a special function in 4GL (called *web service stub*) which will invoke the web service operation. In Lycia Studio both the meta data file and the client stub are created automatically.

Finding a URI of an External Web Service

As has already been stated, web services are published in special catalogues in the Internet. You can search for a required service there and find its *URI (Uniform Resource Identifier)* in order to connect to the server via the Internet. A URI represents the location of the WSDL schema – the web service definition file. A URI includes the name of an open Internet protocol (most often HTTP), the name of the server on which the web service resides, the name of the web service followed by the question mark and the acronym WSDL which stands for *Web Service Description Language*.

<protocol_name>://<web_server_name>/<web_service_name>?WSDL

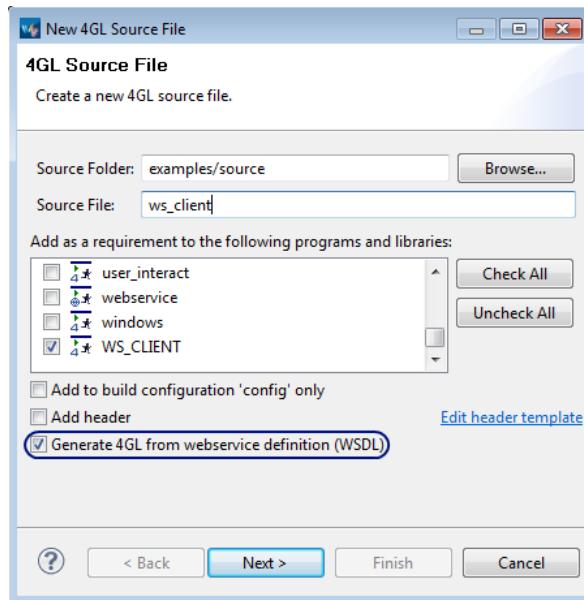
For example, the following address specifies that the WSDL schema of the web service “length.asmx” can on the server called “webservice.net” be accessed via the HTTP protocol.

<http://www.webservicex.net/length.asmx?WSDL>

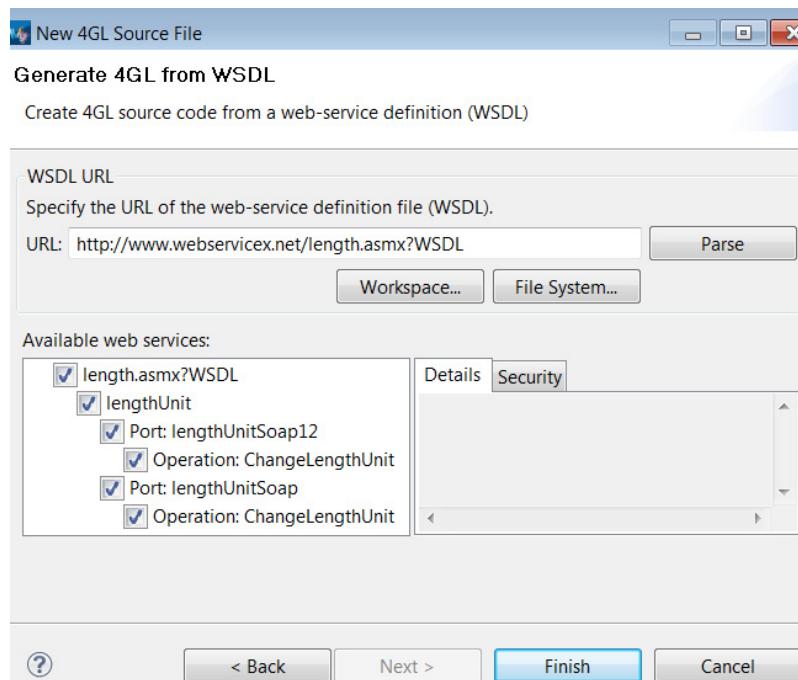
Generating a Meta Data Description File and Web Service Client Stub

Once you know the URI of a particular web service, the next thing you should do is generate the web service meta data definition file and the 4GL web service stub. To generate it, you should do the following:

1. Create a new 4GL program in Lycia Studio in the standard way
2. Create a new 4GL source file for this program.
 - a. When the new file creation wizard opens, you will see the following option at its bottom: “Generate 4GL from webservice definition (WSDL)”. Check this option.

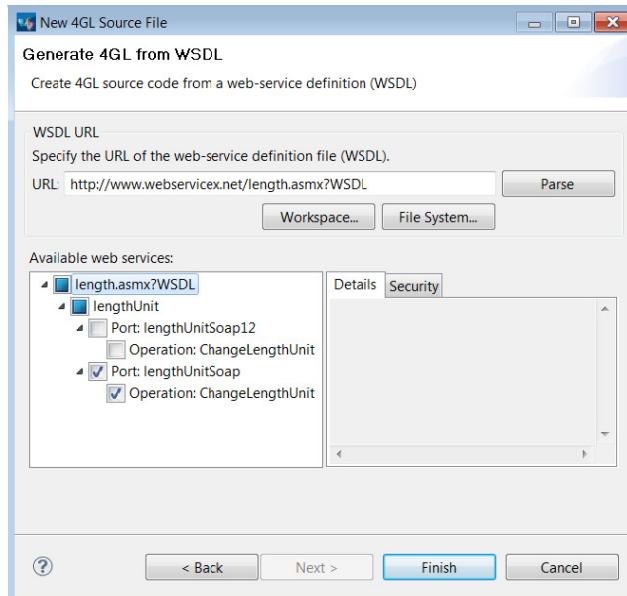


- b. Specify a name for the new file and press the **Next >** button.
3. You will be presented with another wizard prompting you to provide the URI of the required web service in the format described above:
4. Click **Parse** button. In the pane “Available web services” you will see the list of web services you can use, operations (i.e. functions) they perform, and the ports by which they can be accessed. As the following screenshot demonstrates, Lycia Studio offers two ports for the messages to be transported: SOAP 1.1. and SOAP 1.2.



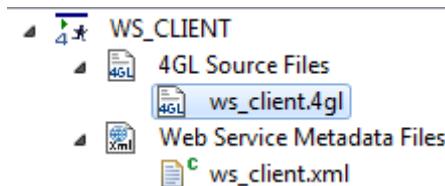


5. You can choose to retain both of these ports, or select only one of them. In the second case, you should uncheck the port you don't need, as follows:



6. Click **Finish** to create the file.

After pressing the Finish button, the meta data definition file, along with the function which invokes the web service, will be automatically generated:



If you open the definition file (which is the .xml file), you will see the basic information about the web service such as the service name, port name, operation name and so on. The web service stub, in its turn, looks like this:

```
#####
#FUNCTION lengthUnit_lengthUnitSoap_ChangeLengthUnit
#web service stub to invoke the web service operation
#ChangeLengthUnit
#URL          : http://www.webservicex.net/length.asmx
#Binding      : http://www.webservicex.NET/ChangeLengthUnit
#Address Style : SOAP1.1
#####
```



```
FUNCTION lengthUnit_lengthUnitSoap_ChangeLengthUnit(input_record)
RETURNS(output_record)

DEFINE input_record RECORD
    LengthValue FLOAT,
    fromLengthUnit STRING,
    toLengthUnit STRING
END RECORD

DEFINE output_record RECORD
    ChangeLengthUnitResult FLOAT
END RECORD

DEFINE ws WEBSERVICE
    CALL ws.LoadMeta("ws_cl_main2.xml")
    CALL ws.SelectOperation("lengthUnit", "lengthUnitSoap",
                           "ChangeLengthUnit")
    CALL ws.Execute(input_record.*)
    RETURNING output_record.*

    RETURN output_record.*
END FUNCTION
```



Note: It is advised that you restart Tomcat each time before you deploy or redeploy a web-service. This will prevent the deployment from possible mistakes

Explaining the Stub Syntax

The web service stub includes a number of built-in functions used to work with a web service. Let us take a closer look at them. These functions have a specific syntax which generally consists of a variable of the WEBSERVICE data type, the function name and the arguments.

The WEBSERVICE Data Type

First of all, we should say a few words about a specific data type called WEBSERVICE. Thanks to this data type, the developer doesn't have to use web service handles (special identifiers for particular web services) any longer. This makes the interaction with web services much easier, and also requires less code to be written. A WEBSERVICE data type variable can be declared just like any other variable in 4GL:

```
DEFINE variable_name WEBSERVICE
```



The *variable_name* here stands for a unique variable identifier, while the WEBSERVICE keyword indicates that the variable belongs to the WEBSERVICE data type. For example:

```
DEFINE ws WEB SERVICE
```

Loading a Web Service Meta Data Definition File

The first thing to be done for an application to start working with a specific web service is to load its WSDL meta data definition file. This is done with the help of the *LoadMeta()* function. It takes a single argument – a quoted name of the meta data definition file with the .xml extension.

```
LoadMeta( "metadata_file_name.xml" )
```

To use it with a WEBSERVICE data type variable you should join the variable with the function name using the *period* operator “.”

```
CALL variable_name.LoadMeta( "meta_data_file_name.xml" )
```

The code below will load the meta data file of the web service *ws* with the help of the *LoadMeta ()* function.

```
CALL ws.LoadMeta ( "ws_cl_main2.xml" )
```

Selecting a Web Service Operation

After the meta data file has been loaded, you should select a web service operation you want to invoke. You can do it by calling the *SelectOperation()* function.

```
SelectOperation( "service_name" , "port_name" , "operation_name" )
```

The function takes three arguments. The first argument is the name of the web service currently in use. The second argument is the port name by which the operation can be accessed, and the third – the name of the operation itself. All the arguments must be quoted and separated by commas.

Using the same variable of the WEBSERVICE data type, we invoke an operation called ChangeLengthUnit for the same WEBSERVICE variable *ws*.

```
CALL ws.SelectOperation ( "LengthUnit" , "LengthUnitSoap" ,  
"ChangeLengthUnit" )
```

Executing a Web Service Operation

Once the operation is selected, you can execute it. This is done by means of the *Execute()* function.

```
Execute([parameters]) [RETURNING parameters]
```

The argument this function accepts is optional, and it represents the parameters passed to the stub from the calling routine. If no parameters are passed, they are replaced by NULL values. There can also be a RETURNING clause which specifies return variables.

```
CALL ws.Execute(input_record.* ) RETURNING output_record.*
```



Making it All Work

To make the program work, you need to create the MAIN program block next. You can do it in the same file to which the web service stub was added, or create a separate file for this purpose.

We will add the following code to the web service stub module preceding the FUNCTION program block which was automatically generated:

```
MAIN

DEFINE
    source_unit, target_unit VARCHAR(255),
    source_length, target_length FLOAT,
    inp_char CHAR

    LET source_unit = "Miles"
    LET target_unit = "Meters"
    LET source_length = 10

    DISPLAY "Testing the web service operation ",
        "ChangeLengthUnit(input_record)" AT 3,5

    DISPLAY "source_unit =", source_unit AT 5,5
    DISPLAY "target_unit =", target_unit AT 6,5
    DISPLAY "source_length =", source_length AT 7,5

    DISPLAY "CALL ChangeLengthUnit(",
        source_length, ", ", source_unit, ", ", target_unit, ")
        RETURNING target_length" AT 9,5

    CALL ChangeLengthUnit(source_length, source_unit, target_unit)
    RETURNING target_length

    DISPLAY "Returned value=", target_length AT 11,5
    PROMPT "Press any key to close this application" FOR CHAR inp_char

END MAIN
```

Note that the name of the operation in the web service stub is preceded by the service name and the name of the port separated by underscores (lengthUnit_lengthUnitSoap_ChangeLengthUnit):



```
FUNCTION lengthUnit_lengthUnitSoap_ChangeLengthUnit(input_record)
RETURNS(output_record)
```

The function description in the stub file also contains the information about the variables that are returned by this function (DEFINE output_record RECORD...). The order in which these variables are listed does not necessarily correspond to the order in which the variables are listed in the web function itself. However, the order of the variables in the RETURNING clause of the CALL statement should correspond to the variables order in the stub file.

However, the name of the function in the main block when it is called refers to the web service operation *ChangeLengthUnit* only:

```
CALL ChangeLengthUnit(source_length, source_unit, target_unit)
RETURNING target_length
```

Correspondingly, you should leave only the operation name in the web service stub, thus you need to edit the automatically generated function to achieve the following result:

```
FUNCTION ChangeLengthUnit(input_record) RETURNS(output_record)
```

On the other hand you can use the long function name in the MAIN section, then you will not need to edit the automatically generated function. The main point here is that the name of the function in its declaration and in the calling routine should be the same.

Now, if you build and run the program, you will get the following result:

```
ws_prog.exe -d informix
Accept Cancel Help
Press any key to close this application

Testing the web service operation ChangeLengthUnit(input_record)

source_unit =Miles
target_unit =Meters
source_length =10.0

CALL ChangeLengthUnit(10.0,Miles,Meters) RETURNING target_length

Returned value=16093.44
```

Example

The following example demonstrates how a web service can be created and used by 4gl applications. This may be called an extended version of the example described above in this chapter.

Before you start applying this example, install a web server, if you don't have access to one. You can use Tomcat which can be downloaded here: <http://tomcat.apache.org/>. For the details about how to set up Tomcat for working with web services see the Tomcat documentation and "Webservices Getting Started" guide which is shipped together with Lycia package or can be downloaded at our web site www.querix.com.



You also should have Microsoft Visual Studio installed.

Web Service

Here is the text of the web service to which the main program will refer. The web service is called 'units_converter' and contains four web functions. You need to create an object called Web Service instead of a 4GL program and add this file there and compile the web service.

```
#####
# This function is used to convert the units of length
#####
WEB FUNCTION convert_length(unit, target_unit, value) RETURNS (result)
DEFINE unit, target_unit CHAR(20),
       value, result DECIMAL (15,5),
       rate INTEGER

IF target_unit = "Centimetres"
  THEN LET rate = 100
  ELSE LET rate = 1
END IF

CASE unit
  WHEN "Mile"
    LET result = value*rate*1609.344
  WHEN "Yard"
    LET result = value*rate*0.9144
  WHEN "Foot"
    LET result = value*rate*0.3048
  WHEN "Inch"
    LET result = value*rate*0.0254
END CASE

# Although the function has no RETURN statement, the value of the
# variable 'result' will be passed to the calling routine because
# the RETURNS clause of the FUNCTION statement contains an unambiguous
# identifier of the variable to be passed
END FUNCTION

#####
# This function is used to convert the units of weight
#####
WEB FUNCTION convert_weight(unit, target_unit, value) RETURNS (result,rate)
DEFINE unit, target_unit CHAR(20),
       value, result DECIMAL (15,5),
       rate INTEGER

IF target_unit = "Grams"
  THEN LET rate = 1000
  ELSE LET rate = 1
END IF

CASE unit
  WHEN "Stone"
    LET result = value*rate*6.35
```



```
WHEN "Pound"
    LET result = value*rate*0.45359
WHEN "Long ton"
    LET result = value*rate*1016.05
END CASE
# The RETURN section contains only one variable. So, the value of this
# variable will be passed to the calling routine, and the value of the
# rate variable will not
RETURN result
END FUNCTION

#####
# This function is used to convert the units of area
#####
WEB FUNCTION convert_area(unit, target_unit, value) RETURNS (result)
DEFINE unit, target_unit CHAR(20),
       value, result DECIMAL (15,5),
       rate INTEGER

IF target_unit = "Square Metres"
    THEN LET rate = 1000000
    ELSE LET rate = 1
END IF

CASE unit
    WHEN "Square mile"
        LET result = value*rate*2.59
    WHEN "Acre"
        LET result = value*rate*0.00404686
    WHEN "Rood"
        LET result = value*rate*0.00101171
END CASE

RETURN result
END FUNCTION

#####
# This function is used to fill the form combo boxes with values
# depending on the type of the measurement units chosen by the user
#####
# This function returns several values including two arrays

WEB FUNCTION fill_lists(un_type) RETURNS(mes_units, target_units, i)

DEFINE un_type CHAR(20),
      i      INTEGER
DEFINE mes_units DYNAMIC ARRAY [1] OF CHAR (20)
DEFINE target_units DYNAMIC ARRAY[1] OF CHAR (20)

CASE un_type
    WHEN "Length"

        # These commands insert four values to the dropdown list of the
        # 'unit' combo box
```



```
LET mes_units[1] = "Mile"
LET mes_units[2] = "Yard"
LET mes_units[3] = "Foot"
LET mes_units[4] = "Inch"
LET i = 4
# These commands insert two values to the dropdown list of the
# 'target_unit' combo box
LET target_units[1] = "Metres"
LET target_units[2] = "Centimetres"

WHEN "Weight"
    LET mes_units[1] = "Stone"
    LET mes_units[2] = "Pound"
    LET mes_units[3] = "Long ton"
    LET i = 3

    LET target_units[1] = "Kilograms"
    LET target_units[2] = "Grams"

WHEN "Area"
    LET mes_units[1] = "Square mile"
    LET mes_units[2] = "Acre"
    LET mes_units[3] = "Rood"
    LET i = 3

    LET target_units[1] = "Square Metres"
    LET target_units[2] = "Square Kilometres"

END CASE

RETURN mes_units, target_units, i
END FUNCTION
```

Then you need to deploy it. After you deploy your web service, you will see the following in the web browser at your web service destination page (if you use the Tomcat):

units_converter

Service EPR : http://localhost:8080/axis2/services/units_converter

Service Description : units_converter

Service Status : Active
Available Operations

- fill_lists
- convert_length
- convert_area
- convert_weight



Client

Below is given the source code of the program which references the 'units_converter' web service. It is a normal 4GL program, you should create it as a 4GL Program object in Lycia Studio.

Create a normal 4GL file with the following code within this program:

```
MAIN
DEFINE
unit_type, unit, target_unit CHAR (20),
value, result DECIMAL (15,5),
rate,i, counter INTEGER,
mes_units, target_units DYNAMIC ARRAY [1] OF CHAR (20)

OPEN WINDOW win1 AT 2,2 WITH FORM "converter_form"

DISPLAY "!" TO qit
DISPLAY "!" TO convert
DISPLAY "!" TO cler

OPTIONS INPUT WRAP

INPUT BY NAME unit_type, unit, target_unit, value
AFTER FIELD unit_type
# If the 'unit' and the 'target_unit' fields already have some values
# in the combo box lists, these values have to be deleted
CALL fgl_list_clear ("unit",1,5)
CALL fgl_list_clear ("target_unit",1,2)

CALL ws_fill_lists(unit_type)
RETURNING i, mes_units, target_units

FOR counter = 1 to i
    CALL fgl_list_insert("unit", counter, mes_units[counter])
END FOR

FOR counter = 1 to 2
    CALL fgl_list_insert("target_unit", counter, target_units[counter])
END FOR

ON ACTION (qit)
EXIT PROGRAM

ON ACTION (cler)
CLEAR FORM
DISPLAY "!" TO qit
DISPLAY "!" TO convert
DISPLAY "!" TO cler

ON ACTION (convert)
CASE unit_type
    # Below, we use the names of the web functions that we changed
    WHEN "Length" CALL ws_convert_length(unit, target_unit, value) RETURNING
result
```



```
# the following function has two variables in the RETURNING clause, however
# only the variable result will get its value, because the variable rate
# is not specified in the RETURN section of the web function convert_length
    WHEN "Weight" CALL ws_convert_weight(unit, target_unit, value) RETURNING
rate, result

    WHEN "Area"     CALL ws_convert_area   (unit, target_unit, value) RETURNING
result
END CASE
DISPLAY BY NAME result
END INPUT

DISPLAY "!" TO convert

END MAIN
```

The Form File

Here is the form file 'converter_form.per' to where the input is performed:

DATABASE formonly

```
SCREEN {

\gp-----q\g
\g| [101 ] |\g
\g| |\g
\g| |\g
\g| \gTypes of units \g[f01 ] \gInitial value: \g [f04 ]|\g
\g| |\g
\g| \gInput units \g[f02 ] \gConverted value: \g[f05 ]|\g
\g| |\g
\g| \gTarget units \g[f03 ] |\g
\g| |\g
\g| |\g
\g| [b01 ] [b02 ] [b03 ] |\g
\g| |\g
\g| |\g
\gb-----d\g
}
```

ATTRIBUTES

```
f01 = formonly.unit_type, widget = "combo",
      include = ("Length", "Weight", "Area"), REQUIRED,
      comments = "Choose the type of measurement units";

f02 = formonly.unit, widget = "combo", wordwrap,
      comments = "Choose the measurement unit to convert";

f03 = formonly.target_unit, widget = "combo", wordwrap,
      comments = "Choose the unit to which the value will be converted";

f04 = formonly.value TYPE DECIMAL,
```



```
comments = "Enter the value to be converted (INTEGER)";

f05 = formonly.result TYPE DECIMAL, noentry, color = reverse bold;

b01 = formonly.convert, widget = "button", config = "convert {Convert}";
b02 = formonly.cler, widget = "button", config = "cler {Clear}";
b03 = formonly.qit, widget = "button", config = "qit {Quit}";

101 = formonly.headr, widget = "label",
      config = "Enter the units to convert", color = magenta bold, center;
```

The Stub File

Then create another 4GL file for the same program. Do not forget to check the "Generate 4GL from webservice definition" option when creating the 4GL file and use the URL of the web service you've just created and deployed as reference. Click on the web service in your browser to get its URL

Here is the stub file 'web_functions' that is generated from WSDL. The file contains the descriptions of the available web functions. When creating the stub file, we decided to use only port 11. We also changed the automatically generated function names to shorter ones to make them more convenient to refer to. The initial function names are given in the commented parts which precede each function description. It is important to make sure that the function names in the calling routine match the function names in the stub file. It is also advised that you make sure that the order of arguments in the calling routine matches the order of the values to be returned specified in the stub file.

You SHOULD NOT copy and paste this text and the following one to your program. They must be generated automatically by the Lycia tool for creation stub files. You can compare the texts given below to those you get in your stub files.

```
#####
#FUNCTION units_converter_units_converterHttpSoap11Endpoint_convert_weight
#
#web service stub to invoke the web service operation convert_weight
#
#URL:
http://localhost:8080/axis2/services/units_converter.units_converterHttpSoap11Endpoint/
#Binding        : document
#Address Style  : SOAP1.1
#
#####

# We renamed the name of the function to make it more convenient to refer to
FUNCTION ws_convert_weight(input_record) RETURNS(output_record)
  DEFINE input_record RECORD
    args0 STRING,
    args1 STRING,
    args2 DECIMAL(32)
  END RECORD

  DEFINE output_record RECORD
    return RECORD
      rate INTEGER,
      result DECIMAL(32)
```



```
END RECORD
END RECORD

DEFINE ws WEBSERVICE

CALL ws.LoadMeta( "web_functions.xml" )
CALL ws.SelectOperation( "units_converter",
    "units_converterHttpSoap11Endpoint", "convert_weight" )
CALL ws.Execute(input_record.* ) RETURNING output_record.*

RETURN output_record.*
END FUNCTION

#####
#FUNCTION units_converter_units_converterHttpSoap11Endpoint_fill_lists
#
#web service stub to invoke the web service operation fill_lists
#
#URL:
http://localhost:8080/axis2/services/units_converter.units_converterHttpSoap11Endpoint/
#Binding      : document
#Address Style : SOAP1.1
#
#####
# We renamed the name of the function to make it more convenient to refer to
FUNCTION ws_fill_lists(input_record) RETURNS(output_record)
    DEFINE input_record RECORD
        args0 STRING
    END RECORD

    DEFINE output_record RECORD
        return RECORD
            i INTEGER,
            mes_units DYNAMIC ARRAY WITH 1 DIMENSIONS OF STRING,
            target_units DYNAMIC ARRAY WITH 1 DIMENSIONS OF STRING
        END RECORD
    END RECORD

    DEFINE ws WEBSERVICE

    CALL ws.LoadMeta( "web_functions.xml" )
    CALL ws.SelectOperation( "units_converter",
    "units_converterHttpSoap11Endpoint", "fill_lists" )
    CALL ws.Execute(input_record.* ) RETURNING output_record.*

    RETURN output_record.*
END FUNCTION

#####

#FUNCTION units_converter_units_converterHttpSoap11Endpoint_convert_length
#
#web service stub to invoke the web service operation convert_length
#
```



```
#URL:  
http://localhost:8080/axis2/services/units_converter.units_converterHttpSoap11Endpoint/  
#Binding      : document  
#Address Style : SOAP1.1  
#  
#####  
# We renamed the name of the function to make it more convenient to refer to  
FUNCTION ws_convert_length(input_record) RETURNS(output_record)  
    DEFINE input_record RECORD  
        args0 STRING,  
        args1 STRING,  
        args2 DECIMAL(32)  
    END RECORD  
  
    DEFINE output_record RECORD  
        return DECIMAL(32)  
    END RECORD  
  
    DEFINE ws WEBSERVICE  
  
        CALL ws.LoadMeta( "web_functions.xml" )  
        CALL ws.SelectOperation( "units_converter",  
            "units_converterHttpSoap11Endpoint", "convert_length" )  
        CALL ws.Execute(input_record.* ) RETURNING output_record.*  
  
        RETURN output_record.*  
    END FUNCTION  
  
#####  
#FUNCTION units_converter_units_converterHttpSoap11Endpoint_convert_area  
#  
#web service stub to invoke the web service operation convert_area  
#  
#URL:  
http://localhost:8080/axis2/services/units_converter.units_converterHttpSoap11Endpoint/  
#Binding      : document  
#Address Style : SOAP1.1  
#  
#####  
# We renamed the name of the function to make it more convenient to refer to  
FUNCTION ws_convert_area(input_record) RETURNS(output_record)  
    DEFINE input_record RECORD  
        args0 STRING,  
        args1 STRING,  
        args2 DECIMAL(32)  
    END RECORD  
  
    DEFINE output_record RECORD  
        return DECIMAL(32)  
    END RECORD  
  
    DEFINE ws WEBSERVICE  
  
        CALL ws.LoadMeta( "web_functions.xml" )
```



```
CALL ws.SelectOperation( "units_converter",
"units_converterHttpSoap11Endpoint", "convert_area")
CALL ws.Execute(input_record.*) RETURNING output_record.*

RETURN output_record.*
END FUNCTION
```

The XML File

The xml file is generated together with the stub by Lycia Studio automatically. It can contain the following text (it can be written in one line):

```
<?xml version="1.0" encoding="UTF-8"
standalone="no"?><wsdl><version>1.1</version><targetNameSpace>http://wsoutput.querix.com</target
NameSpace><service><serviceName>units_converter</serviceName><port><portName>units_converterH
tpSoap11Endpoint</portName><url>http://localhost:8080/axis2/services/units_converter.units_converterHt
pSoap11Endpoint</url><SOAPVersion>SOAP1.1</SOAPVersion><bindingStyle>document</bindingStyle>
<operation><operationName>convert_area</operationName><MEP>SendAndReceive</MEP><SoapAction
>urn:convert_area</SoapAction><bindingStyle>document</bindingStyle><messageNamespace>http://ws
output.querix.com</messageNamespace><InputMessage><messageName>convert_areaRequest</messag
eName><variable><variableName>args0</variableName><targetNameSpace>http://wsoutput.querix.com
</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNames
pace><variableType
baseType="string">string</variableType><variableDimension/><variableNillable>true</variableNillable></
variable><variable><variableName>args1</variableName><targetNameSpace>http://wsoutput.querix.com
</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNames
pace><variableType
baseType="string">string</variableType><variableDimension/><variableNillable>true</variableNillable></
variable><variable><variableName>args2</variableName><targetNameSpace>http://wsoutput.querix.com
</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNames
pace><variableType
baseType="decimal">decimal</variableType><variableDimension/><variableNillable>true</variableNillable
></variable></InputMessage><OutputMessage><messageName>convert_areaResponse</messageName>
<variable><variableName>return</variableName><targetNameSpace>http://wsoutput.querix.com</target
NameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNamespac
e><variableType
baseType="decimal">decimal</variableType><variableDimension/><variableNillable>true</variableNillable
></variable></OutputMessage></operation><operation><operationName>convert_length</operationNa
me><MEP>SendAndReceive</MEP><SoapAction>urn:convert_length</SoapAction><bindingStyle>document
</bindingStyle><messageNamespace>http://wsoutput.querix.com</messageNamespace><InputMessage>
<messageName>convert_lengthRequest</messageName><variable><variableName>args0</variableName
><targetNameSpace>http://wsoutput.querix.com</targetNameSpace><variableTypeNamespace>http://ww
w.w3.org/2001/XMLSchema</variableTypeNamespace><variableType
baseType="string">string</variableType><variableDimension/><variableNillable>true</variableNillable></
variable><variable><variableName>args1</variableName><targetNameSpace>http://wsoutput.querix.com
</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNames
pace><variableType
baseType="string">string</variableType><variableDimension/><variableNillable>true</variableNillable></
variable><variable><variableName>args2</variableName><targetNameSpace>http://wsoutput.querix.com
</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNames
pace><variableType
baseType="decimal">decimal</variableType><variableDimension/><variableNillable>true</variableNillable
```



```
></variable></InputMessage><OutputMessage><messageName>convert_lengthResponse</messageName><variable><variableName>return</variableName><targetNameSpace>http://wsoutput.querix.com</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNamespace><variableType>decimal</variableType><variableDimension/><variableNillable>true</variableNillable></variable></OutputMessage></operation><operation><operationName>convert_weight</operationName><MEP>SendAndReceive</MEP><SoapAction>urn:convert_weight</SoapAction><bindingStyle>document</bindingStyle><messageNamespace>http://wsoutput.querix.com</messageNamespace><InputMessage><messageName>convert_weightRequest</messageName><variable><variableName>args0</variableName><targetNameSpace>http://wsoutput.querix.com</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNamespace><variableType>string</variableType><variableDimension/><variableNillable>true</variableNillable></variable><variable><variableName>args1</variableName><targetNameSpace>http://wsoutput.querix.com</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNamespace><variableType>string</variableType><variableDimension/><variableNillable>true</variableNillable></variable><variable><variableName>args2</variableName><targetNameSpace>http://wsoutput.querix.com</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNamespace><variableType>decimal</variableType><variableDimension/><variableNillable>true</variableNillable></variable></InputMessage><OutputMessage><messageName>convert_weightResponse</messageName><variable><variableName>return</variableName><targetNameSpace>http://wsoutput.querix.com</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNamespace><variableType>units_converter_convert_weight_implicit</variableType><variableDimension/><variableNillable>true</variableNillable><members><variable><variableName>rate</variableName><targetNameSpace>http://wsoutput.querix.com/xsd</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNamespace><variableType>int</variableType><variableDimension/><variableNillable>false</variableNillable></variable><variable><variableName>result</variableName><targetNameSpace>http://wsoutput.querix.com/xsd</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNamespace><variableType>decimal</variableType><variableDimension/><variableNillable>true</variableNillable></variable></members></variable></OutputMessage></operation><operation><operationName>fill_lists</operationName><MEP>SendAndReceive</MEP><SoapAction>urn:fill_lists</SoapAction><bindingStyle>document</bindingStyle><messageNamespace>http://wsoutput.querix.com</messageNamespace><InputMessage><messageName>fill_listsRequest</messageName><variable><variableName>args0</variableName><targetNameSpace>http://wsoutput.querix.com</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNamespace><variableType>string</variableType><variableDimension/><variableNillable>true</variableNillable></variable></InputMessage><OutputMessage><messageName>fill_listsResponse</messageName><variable><variableName>return</variableName><targetNameSpace>http://wsoutput.querix.com</targetNameSpace><variableTypeNamespace>http://wsoutput.querix.com/xsd</variableTypeNamespace><variableType>units_converter_fill_lists_implicit</variableType><variableDimension/><variableNillable>true</variableNillable><members><variable><variableName>i</variableName><targetNameSpace>http://wsoutput.querix.com/xsd</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNamespace><variableType>int</variableType><variableDimension/><variableNillable>false</variableNillable></variable><variable><variableName>mes_units</variableName><targetNameSpace>http://wsoutput.querix.com/xsd</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNamespace><variableType>
```



```
baseType="string">string</variableType><variableDimension>[]</variableDimension><variableNillable>true</variableNillable></variable><variable><variableName>target_units</variableName><targetNameSpace>http://wsoutput.querix.com/xsd</targetNameSpace><variableTypeNamespace>http://www.w3.org/2001/XMLSchema</variableTypeNamespace><variableType>baseType="string">string</variableType><variableDimension>[]</variableDimension><variableNillable>true</variableNillable></variable></members></variable></OutputMessage></operation></port></service></wsdl>
```



The Notion of Reports

We know that the ability to retrieve and output information from the database is very important. A report is the most convenient and effective way to perform output from a database. However, a report can perform output of any information passed by the program and not only of database data.

The General Notion of Report

To create a report in 4GL you need to create a REPORT program block and a report driver, which will invoke the report described in the REPORT program block. The report driver invokes a report in a way similar to the way the CALL statement invokes a function. However, a report driver has a more complicated structure than the CALL statement.

The REPORT program block is similar to the MAIN and FUNCTION program blocks and is used to arrange and format the values sent to the report. However, the values themselves are sent to the REPORT program block from the routine which invoked the report with the help of the report driver. A REPORT program block cannot process and send to the output destination anything which was not passed to it by the report driver.

4GL reports have a number of useful and convenient features which are listed below:

- A report can be displayed to the screen and be edited by the user;
- The programmer has the full control over the report page layout; they can specify the peculiarities of columnar presentation, define first and the last page headers, page trailers and headings, and set special formatting before and after sorted groups of values are outputted.
- The report syntax supports a set of control blocks that are used to manipulate the data retrieved from a database.
- A report control block can include a set of functions that calculate and display sums, averages, maximum and minimum values, frequencies, etc.
- A report can include the rows retrieved both from a cursor and input records retrieved by any other source (e.g., you can perform an output from one or several SELECT statements or from the user input).
- 4GL gives the programmer an ability to update the database or execute any sequence of 4GL or SQL statements when the report is being created. For example, you can make the program create and display a message that contains a second report while processing the first one.

The REPORT Program Block

A report is specified within the REPORT program block, which starts with the REPORT statement and ends with the END REPORT keywords. The REPORT program block can include the report arguments, format, output settings, etc.

The report specified in the REPORT program block can be executed from the MAIN control block, or from any FUNCTION or another REPORT program block with the help of a report driver. However, a REPORT block cannot be specified within any of the other program blocks.

The generalized structure of the REPORT program block is:

```
REPORT report_name (argument_list)
```



```
[DEFINE clause]
[OUTPUT clause]
[ORDER BY clause]
FORMAT clause
END REPORT
```

The *report_name* is the 4GL identifier of the report. The report driver will reference the report using the report name. The requirements for the report name are the same as those for other identifiers used in 4GL (i.e. it must be unique among other report names in the same program).

The *argument list* is one or more names of formal arguments for each input record separated by commas. The arguments are represented by the local variables; they store values that are sent to the report during the report driver processing. These variables can be of RECORD data type, but you cannot specify an argument as *record.**. You should use just the name of the program record, without a point and an asterisk (e.g., *REPORT my_first (prog_rec)*). You cannot use an ARRAY value as an argument.

All the variables that are used as arguments in the REPORT block including the report arguments must be declared within this block.

A Simplest Case of the REPORT Program Block

The REPORT program block, starting with REPORT and ending with FINISH REPORT keywords, including all the options and statements between them, is also called a *report definition*. The report definition specifies the main features of the report that is to be processed and the format of the result that will be presented to the user.

The report definition must include the FORMAT section in order to make the program process the report. All the other clauses are optional. The FORMAT section defines what data is to be sent to the report output and in which format.

The DEFINE section is not obligatory, if the report does not have local variables and its arguments are declared as module or global variables. However, it is advisable that the report arguments are of local scope. Thus the DEFINE section should also be present. Its syntax is like that of a regular DEFINE statement.

The Default Report Format

The FORMAT section can have a special form which specifies that the report output is the default one without any additional formatting.

The FORMAT EVERY ROW form of the FORMAT section specifies the default report format. If the program encounters these keywords, the report will contain all the values that are passed to the REPORT program block in the order they are passed there, and they will be passed to the report output automatically.

The EVERY ROW option disables all the other statements or control blocks specified within the FORMAT section.

The following report definition contains the EVERY ROW option.

```
REPORT my_first (arg)
DEFINE arg LIKE arg.*
```



FORMAT EVERY ROW

END REPORT

This report receives data from all the columns and rows of the table *custom*. The subsequent examples of the source code used in this chapter will refer to this report.

If you include the EVERY ROW option, the program uses the report arguments as column headings when processing a report. If all fields comprising each input record fit horizontally on a single line, the report prints these headings at the top of each page and the report output itself is printed below them. The result of such report execution may look as follows:

ID	First_name	Last_name	Age
01	Ann	Smith	18
02	Josh	Luwiston	32
03	Terry	Jones	24
04	Nicolas	Space	28
05	Jouliette	Black	35

If the names of the columns do not fit one line, they are placed down the left side of the page and the values of the columns follow them to the right:

ID	05
First_name	Jouliette
Last_name	Black
Age	35
Home_town	London
Address	Golden Ln, 123
ZIP_code	N5
Telephone	
Bank	Royal
Rescent_order	13.08.2010
Destination	London, Golden Ln, 123

Report Driver

A *report driver* is a part of the source code which invokes the report, retrieves the data from a data base or from any other source, and sends them to the REPORT program block, where they are formatted and sent to the output destination. A report driver can be located within the MAIN or any FUNCTION or REPORT program block.



A report driver must include the following elements in the given order in order to invoke the report and make it output the data:

- START REPORT statement.
- A conditional loop (FOREACH, WHILE, FOR)
- OUTPUT TO REPORT statement
- The keywords that close the conditional loop (END FOREACH, END WHILE, END FOR)
- FINISH REPORT or TERMINATE REPORT keywords

The START REPORT statement is used to make the program start processing the specified report.

A conditional loop used within the report driver makes the report output the values send to it more than once. The program adds the rows to the report while the condition is satisfied. If you include no conditional loop to the report driver, the report will consist of only one row.

The OUTPUT TO REPORT statement is used to send the data to the report where they are processed and outputted.

The FINISH REPORT and TERMINATE REPORT keywords are used to complete the report processing.

You cannot activate one and the same REPORT block twice during the program execution. If you use the START REPORT statement to activate the report that has already been activated, the program reinitializes the report, and the output results can be unpredictable. To execute the report control blocks, you have to include the OUTPUT TO REPORT statement to the report driver. Otherwise, the control blocks won't be executed, even if you have specified the START REPORT and the FINISH report statements.

The START REPORT Statement

The START REPORT statement is used to begin processing of a 4GL report and to specify output dimensions and destination.

The syntax of the START REPORT statement is as follows:

```
START REPORT report_name [TO clause] [WITH page_dimensions]
```

The *page dimensions* specify the parameters of the report pages.

When the program activates the START REPORT statement, it performs the following actions:

- Identifies a REPORT block with the specified name by which the input records will be formatted;
- Specifies page delimiters and destination for the output produced by the report;
- Initializes page headers specified within the FORMAT section of the report, if any.

If the default page delimiters and destination satisfy your report requirements, you don't have to include the WITH clause to the report driver.

The START REPORT statement is usually followed by a conditional loop containing the OUTPUT TO REPORT statement which sends the database records to the report.



The TO Clause

The TO clause is used to specify the output destination for the report. By default, if the TO clause is omitted, the report output is performed to the screen.

Below is given an extended syntax of the TO clause:

```
TO SCREEN | PRINTER | FILE filename | PIPE program [IN FORM|IN  
LINE MODE]
```

Thus you can perform output to four different output destinations. You cannot specify more than one report output destination in one TO clause.

The *filename* stands for a quoted string or a character variable containing such string, which specifies a file to where the report results will be stored. The file need not be created beforehand, it is created when the report is processed. If you specify the same file for another report output, or process the same report for the second time, the contents of the file will be replaced by the new report, unless you use the APPEND keyword before the TO FILE keywords.

The *program* stands for a quoted string or a character variable containing such string, which specifies a program or a command line to where the report results will be passed.

If you omit the TO clause when specifying the START REPORT statement, 4GL will send the report output to the destination specified in the REPORT definition. If no definition is specified within either START REPORT or REPORT statements, the program automatically sends the report output to the SCREEN.

If the OUTPUT TO REPORT statement results in an empty set of data records, the TO clause has no effect, because no output is produced, even if formatting control blocks like headers, footers, etc. are specified within the report definition.

You can use the TO clause to specify the following destinations for the report output:

- The screen
- A printer
- A file
- Another program or the command line.

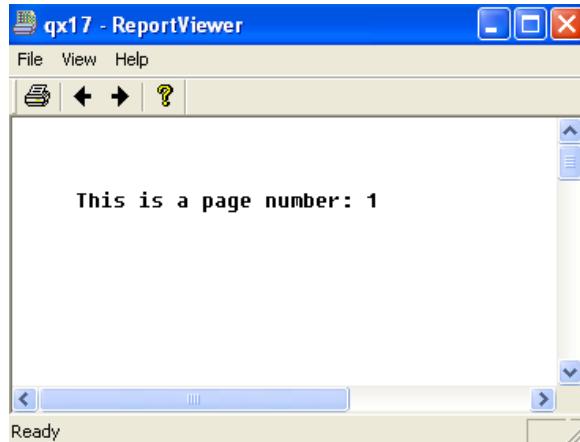
Each of these options is described in details below.

The SCREEN Option

The TO SCREEN option is used to send the report output to the Report window:

```
START REPORT my_first TO SCREEN
```

Querix has designed a Report Viewer - a tool which is a part of graphical clients. When the program is run in a GUI mode and the report is passed to the screen, the client uses this tool to display the report results. The Report Viewer contains the complete report; the user can page through the report by pressing the left and right arrow buttons on the toolbar of the Report Viewer. Below is given a screenshot of the Report Viewer:



If the application is run in the character mode, the report output will be sent to the Notepad file which will be opened automatically. All the pages of the report will go one by one in one and the same file and will be available at the same time.

The FILE Option

You can use the TO FILE *filename* option to send the report output to the specified file. The *filename* can include only the name of the destination file, if it is located in the same folder as the executed program file. If the executed program file and the destination file are located in different folders, the *filename* should include the path. The filename shouldn't include the file extension. The following line sends the output from the report *my_first* to the file named *report_f*.

```
START REPORT my_first TO "report_f"
```

You don't have to create a destination file before you specify it in the TO FILE clause; the program automatically creates a file which has the name specified in the quotes. If a file with such a name already exists, the program will overwrite it unless the APPEND keyword is specified before the TO FILE keywords. If the APPEND keyword is present, the current report will be appended to the previous one. The APPEND feature works only when both reports are sent to a file, because reports sent to a printer are always appended and it is the natural course of things and reports sent to the screen are not overwritten or appended while they are opened in different instances of the Report Viewer.

A report statement with the APPEND keyword may be specified as follows:

```
START REPORT my_second APPEND TO FILE "report_f"
```

The PIPE Option

You can use the TO PIPE option of the START REPORT statement in order to send the report output to a specified program or command line. The character string or the variable specified for the TO PIPE option can include command line arguments as well as the path to the program. The following example demonstrates how the output can be piped from the report to the program named *others*:

```
START REPORT my_first TO PIPE "addon/others"
```



The TO clause of the report driver can be used to specify whether the program should operate in line mode or in formatted mode when sending the report output to a pipe. These modes are applicable only if the program runs on a terminal, thus they have no influence on a program running on a normal computer. These modes are described in more detail in the “Running commands” chapter.

The PRINTER Option

The TO PRINTER option is used to send the report output to the device or program which is specified as the value of the DBPRINT environment variable. If you do not specify any specific value to the DBPRINT variable, the report output will be sent to the system default printer:

```
START REPORT my_first TO PRINTER
```

If you want the program to perform the report output to a printer which is not the default one, you have to perform the following actions:

- Set DBPRINT to the necessary value and use the TO PRINTER option of the START REPORT statement.
- Use the TO FILE “filename” or TO “filename” option, to store the report output into the file and then print the file.
- Use the TO PIPE “program” option to pass the output to a command line to print the report output directly. You can also pass the record output to a text editor where you can edit the result before you print it.

The methods of printer specification by means of the DBPRINT variable are described in the “IBM Informix 4GL Reference Manual” document.

The WITH Clause

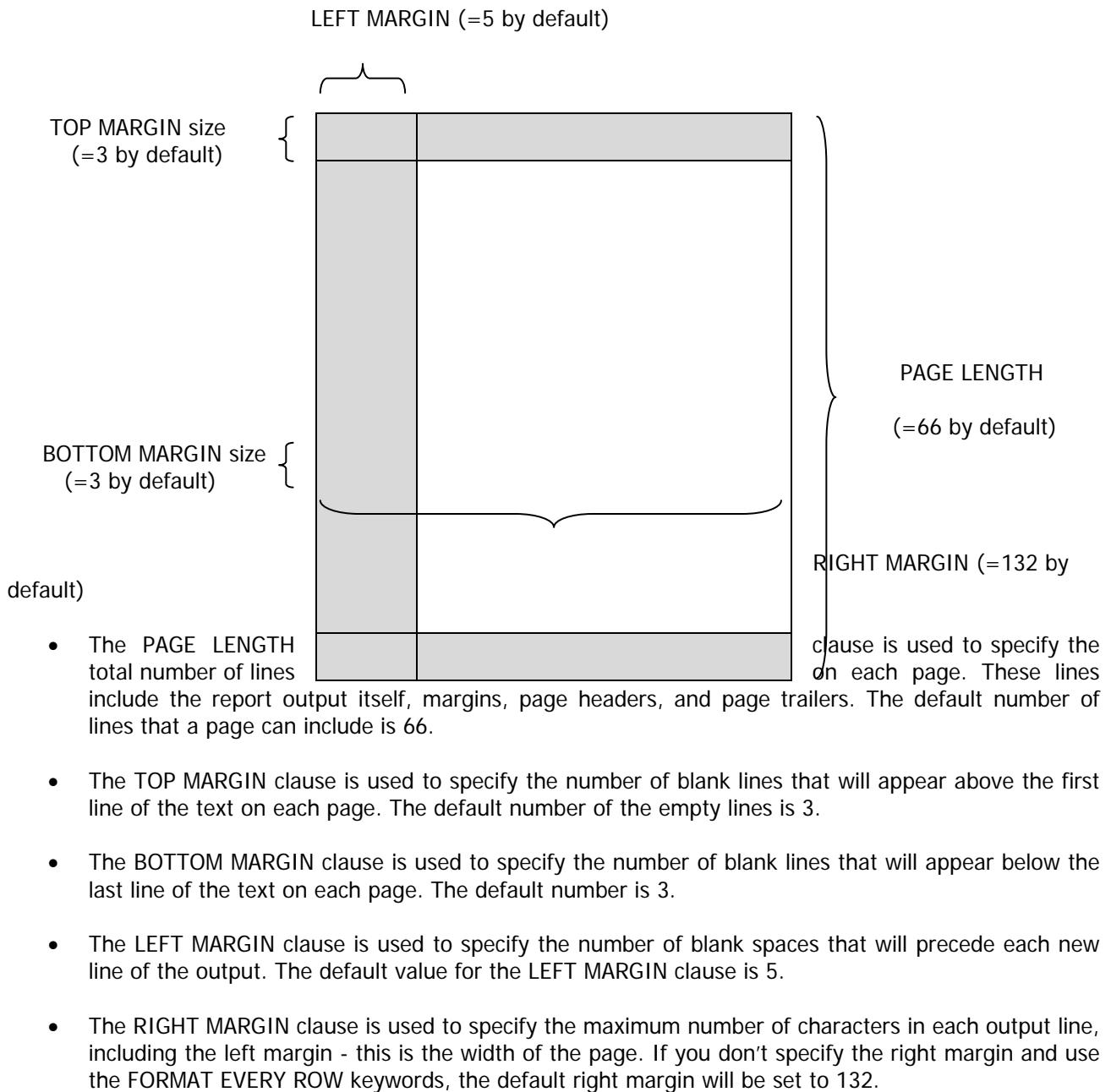
The WITH clause is used to set the dimensions of each page of the report output. The following parameters can be applied to a report page:

- PAGE LENGTH [=] size
- TOP MARGIN [=] size
- BOTTOM MARGIN [=] size
- LEFT MARGIN [=] size
- RIGHT MARGIN [=] size

Each of these parameters can be specified only once within one REPORT statement. They should be separated by commas, if more than one parameter is specified.

The *size* stands for an integer expression specifying the height (in lines) or width (in characters) of a page of report output. These parameters are called the *margins* of the page.

The picture below represents the scheme of a report output page with all the margins shown:



The following example demonstrates the START REPORT statement with a valid WITH clause:

```
START REPORT myrep TO FILE "my_rep"
    WITH TOP MARGIN 1, LEFT MARGIN 8, RIGHT MARGIN 100, BOTTOM
        MARGIN 1
```



OUTPUT TO REPORT Keywords and Conditional Loops

The OUTPUT TO REPORT statement is used to pass a single set of data values to the REPORT statement. It passes data to the report and makes 4GL process and format the data as the next input record of the report.

When mentioning an *input record*, we mean an ordered set of values returned by the statements listed between the parentheses. The program passes these returned values to the specified report, and they become a part of the input record. It is not necessary, but an input record can have correspondence to a row retrieved from the database or to a 4GL program record.

The syntax of the OUTPUT TO REPORT statement is given below:

```
OUTPUT TO REPORT report_name (argument_list)
```

The report name stands for the name of the 4GL report which becomes the basis for the input record format. This identifier must also appear in the corresponding REPORT statement and a preceding START REPORT statement.

The values specified in the argument list of the OUTPUT TO REPORT statement are passed to the REPORT control block, just as the [function arguments](#) in the CALL statement are passed to the corresponding FUNCTION block.

The members of the input record specified in the expression list must correspond to the elements specified in the formal argument list of the REPORT program block in their data types, number, and order.

The values of all data types are passed by value, except for the values of TEXT and BYTE data types, which are passed by reference. BYTE values cannot be displayed within a report. A string <byte value> appears in output record instead.

The OUTPUT TO REPORT statement is typically used within a conditional loop, such as WHILE, FOR, or FOREACH, so, the program passes only one input record at a time. In the following example, we used the FOREACH loop to pass all the rows of the table to the report:

```
DECLARE cur CURSOR FOR SELECT * FROM custom

START REPORT my_first TO SCREEN
    -- the report output will be displayed to
    -- the Report window, the report page has
    -- default size

FOREACH cur INTO pers.*
    -- the cursor moves through the table and the
    -- values of an active row are sent to the
    -- program record pers.

    OUTPUT TO REPORT my_first(pers.*)
        -- the values of the members of the
        -- record pers are sent to the
        -- output buffer and the cursor goes
        -- to the next row

END FOREACH
```



```
FINISH REPORT my_first
```

If the program doesn't execute the OUTPUT TO REPORT statement for any reason, it also does not execute any control blocks specified within the report definition, even if you have included the START REPORT and FINISH REPORT statements into your report driver.

FINISH REPORT AND TERMINATE REPORT Statements

The FINISH REPORT statement is used to indicate the end of the report driver. When the program encounters this statement, it completes the processing of the report.

The FINISH REPORT statement needs the START REPORT statement to be specified earlier in the text and at least one OUTPUT TO REPORT statement referencing the same report between them.

The syntax of the FINISH REPORT statement is very simple:

```
FINISH REPORT report_name
```

It is the FINISH REPORT statement which sends the data from output buffers to the output destination specified within the report definition or START REPORT statement. If you don't specify any destination, the output will be sent to the Report window.

The FINISH REPORT statement influences the data, database items, and the memory the program uses while processing the report. It closes the Select cursor, so you don't have to use the CLOSE CURSOR statement after the report driver. If the program has created any temporary tables to perform aggregate calculations or to order the data retrieved from the database, these tables are closed as soon as the FINISH REPORT statement is executed.

If the report output contains references to any BYTE or TEXT values, the FINISH REPORT statement releases the memory allocated for them.

The TERMINATE REPORT statement makes the program terminate the report that is being currently executed. If this statement is executed, the program does not complete the normal processing of the report and terminates it at the stage where it is. The terminated report is not output to the report destination, if the TERMINATE REPORT statement is executed.

The syntax of the TERMINATE REPORT statement is as follows:

```
TERMINATE REPORT report_name
```

The TERMINATE REPORT statement deletes all the temporary tables and files that the program created when processing the REPORT statement, but does not format any values referenced by the aggregate functions.

The statement may be useful if some data are not included into the report output due to some reasons, and you are not interested in an incomplete report. In fact, the TERMINATE REPORT statement should appear within a conditional statement and be used to detect an error and to specify the actions that are to be performed if an error is detected. If the report procession produces no error, the FINISH REPORT statement should be used rather than the TERMINATE REPORT statement. The following example demonstrates how a report can be terminated if an error occurs:



```
WHENEVER ANY ERROR CONTINUE -- prevents program termination if any
-- error occurs

DECLARE rep_cur CURSOR FOR SELECT * FROM custom

START REPORT my_first TO "report_1"

FOREACH rep_cur INTO pers./*

OUTPUT TO REPORT my_first(pers.*)

IF status>0 THEN TERMINATE REPORT -- the report is terminated if
-- any error occurs

EXIT FOREACH

END IF

END FOREACH

FINISH REPORT
```

Example

This example illustrates the usage of the report driver with a minimum formatted report. For this example to work correctly you need to have the access to the database specified in the DATABASE statement (you can change its name for the name of the database you have). You need to run two applications from the previous chapters which create the database tables and load them with data.

1. First run the application from [Databases in 4GL](#) chapter and click "Create Tables" menu options
2. Then run the application from [Populating Database Tables](#) chapter and click "Load" menu option.

```
#####
# This application illustrates the usage of the report driver and such
# statements as START REPORT, OUTPUT TO REPORT, FINISH REPORT, TERMINATE REPORT,
# as well as the simplest report format.
#####
DATABASE db_examples

MAIN
DEFINE
    r_country_list RECORD LIKE country_list.*,
    dest_rep      CHAR(120),
    num_key       INTEGER

    # This cursor will be used for fetching the information
    # to be put into the reports
    # You can also put any other information not
    # retrieved from a database into a report
DECLARE c_country_list CURSOR FOR
    SELECT * FROM country_list
    ORDER BY country_name
```



```
MENU "Reports"
COMMAND "To a File"
    "Sends a report to a file each time rewriting it"

CALL fgl_winmessage( "Sending Report",
    "The report will be sent to a file", "info")

# This report driver produces a report and sends it to file
# country_list.rep. This file will be created in the
# directory where report_driver.exe is located.

START REPORT country_list -- the report with the same name is
    -- declared at the end of the module
    -- in the REPORT section
TO FILE "country_list.rep"

# We use the FOREACH loop for the driver
# Without a loop the report would contain only one record
# Each time the loop is iterated, the values are passed to the report
FOREACH c_country_list INTO r_country_list.*
    # we need to specify the name
    # and the values passed to the report
    OUTPUT TO REPORT country_list(r_country_list.*)
END FOREACH

FINISH REPORT country_list -- we must specify the report name of the
    -- report we have started above
    -- to complete it
CALL fgl_winmessage( "Report Sent",
    "Report file country_list.rep was created in the location " ||
    "\nwhich depends on the launch method, because we didn't " ||
    "\nspecify the absolute path for it." ||
    "\n-In character mode it is created in the output folder." ||
    "\n-In GUI mode is created on the application server.", "info")

COMMAND "Appended Report"
    "Sends a report to a file appending and not rewriting it"

CALL fgl_winmessage( "Sending Report",
    "The report will be appended to a file", "info")

START REPORT country_list APPEND TO FILE "country_list_append.rep"

FOREACH c_country_list INTO r_country_list.*
    OUTPUT TO REPORT country_list(r_country_list.*)
END FOREACH

FINISH REPORT country_list

CALL fgl_winmessage( "Report Sent",
    "Report file country_list.rep was created in the locatin " ||
    "\nwhich depends on the launch method, because we didn't " ||
    "\nspecify the absolute path for it." ||
```



```
"\n-In character mode it is created in the output folder." ||  
"\n-In GUI mode is created on the application server.",  
"info")  
  
COMMAND "To Printer"  
        "The report will be sent to the SCREEN using FOREACH loop"  
  
        CALL fgl_winmessage("Sending Report",  
                            "The report will be sent to the printer.", "info")  
  
        START REPORT country_list TO PRINTER  
            FOREACH c_country_list INTO r_country_list.*  
                OUTPUT TO REPORT country_list(r_country_list.*)  
            END FOREACH  
        FINISH REPORT country_list  
  
COMMAND "To Screen"  
        "The report will be sent to the SCREEN using WHILE loop"  
  
        CALL fgl_winmessage("Sending Report",  
                            "The report will be sent to the screen " ||  
                            "\n or, in the GUI mode, to the report viewer.", "info")  
  
        LET dest_rep = "SCREEN" -- we assign the report destination to a  
                                -- variable  
  
        START REPORT country_list  
            TO OUTPUT dest_rep -- and then use this variable as the  
                                -- destination  
  
            OPEN c_country_list  
            WHILE TRUE -- this time we use the WHILE loop in the  
                        -- driver  
                FETCH c_country_list INTO r_country_list.*  
                IF status = NOTFOUND THEN  
                    EXIT WHILE  
                END IF  
                OUTPUT TO REPORT country_list(r_country_list.*)  
            END WHILE  
  
        FINISH REPORT country_list  
  
COMMAND "Exit"  
        EXIT PROGRAM  
  
END MENU  
  
END MAIN  
  
#####  
# This is a simple report section which is processed by the report drivers above  
#####  
REPORT country_list(r_country_list) -- here we declare the name and the values  
                                    -- received from the OUTPUT TO statement of  
                                    -- the report drivers
```



```
DEFINE
    r_country_list RECORD LIKE country_list.* -- like in a function we need to
                                                -- declare the report arguments

FORMAT
    # This means every row of report will contain the following information
    # This statement is repeated each time the loop in the report driver is
    # repeated
    ON EVERY ROW
        PRINT r_country_list.id_country, "      ", r_country_list.code_lit,
              "      ", r_country_list.country_name
END REPORT
```



Formatting Report Output

The ability to create reports to output the information stored in a database would be incomplete if the program didn't support influencing the way the data are selected from the database and passed to the report output. The data selection is controlled by the SELECT statement. The format of the output is specified mostly by the options that the REPORT control block includes.

There are two ways to format the report output. One of them is to specify output option within the report driver, the second is specifying the output options within the report definition.

It is important to remember, that the output options specified within the REPORT definition always override the contradicting options specified within the START REPORT statement (except for the OUTPUT TO option that is overridden by the TO clause of the START REPORT statement).

The START REPORT statement and its clauses have been discussed in the previous chapter, so, we will refer to them only in order to specify some peculiarities and correlations with the REPORT program block.

The REPORT statement can include one or several of the optional clauses that specify the report output format. These are the FORMAT, the DEFINE, the OUTPUT, and the ORDER BY sections. They should appear within the statement in the next order:

```
REPORT report_name (argument_list)
    [DEFINE clause]
    [OUTPUT clause]
    [ORDER BY clause]
    FORMAT clause
END REPORT
```

All these sections are described in this chapter.

The DEFINE Section

The DEFINE section of the REPORT statement is used to declare a data type for each formal argument specified in parentheses after the report name, as well as for the local variables that are used within the current REPORT block.

The syntax of the DEFINE statement used within the REPORT block is similar to that used within the MAIN and FUNCTION program blocks:

```
DEFINE variable [, variable...]
```

In general, the rules and restrictions on the variable identifiers and data type specification are also similar to those actual for other DEFINE statements. However, there is a restriction which is valid only for the DEFINE section of the REPORT program block: report arguments cannot be represented by a variable of the ARRAY data type or by an array member. 4GL does not restrict the data types of local report variables which are not specified as report arguments.

The arguments and the DEFINE section are necessary, if the report satisfies at least one of the following conditions:



- If you create a default report by means of the FORMAT EVERY ROW keywords, each record of the report must get a value.
- If you include the ORDER BY section to your report definition, you must define the values that will serve as sorting parameters.

You can use the LIKE keyword to specify data types implicitly, but you can do this only if there is a DATABASE statement specified at the very beginning of the program module which includes the report definition and only if the columns references by the LIKE clause are permanent ones and not temporary.

The statements included into the REPORT control block can refer to global and module variables, but these can cause problems sometimes. In fact, any reference to the variables which are not local for the current report can result in unexpected values. Thus it is highly recommended that you include the DEFINE section into the report block and use the report local variables.

The FORMAT Section

The FORMAT section is the biggest of all the sections that can be included to the REPORT block and is the only necessary one. The section is used to specify how the report output will look after the report is processed.

The FORMAT section uses the global and module variables as well as the values passed to the report through arguments.

We have already discussed the FORMAT EVERY ROW case of the FORMAT section which does not apply any formatting. The FORMAT section which applies some formatting has the following structure:

```
FORMAT
      AFTER GROUP OF
      BEFORE GROUP OF
      *FIRST PAGE HEADER
      *PAGE HEADER
      *ON EVERY ROW
      *PAGE TRAILER
      *ON LAST ROW
```

The section has to include at least one of these elements. The elements that are marked with an asterisk (*) can appear only once within the section. However, the ON EVERY ROW block should be included into the FORMAT section, otherwise your report will have only one page and the report records will not be processed properly.

If you use the FORMAT EVERY ROW keywords which specify the default report format, the other keywords will be ignored even if they are included to the FORMAT section.

Complex REPORT sections can include one or several control blocks, for example, BEFORE GROUP OF, ON LAST ROW, ON EVERY ROW, etc. Each of these control blocks contains a set of statements that are to be executed when the program is processing the report.

All the elements that can be included into the FORMAT section of the REPORT block, except for the EVERY ROW keywords, are described in details further in this chapter.



FORMAT SECTION Control Blocks

You can use the format section control blocks in order to specify the report structure. Each control block must include one or more statements that are to be executed when the program processes some part of the report. If the report processing produces no data records to output, none of the statements within control blocks are executed.

The table below gives a brief description of the order in which the control blocks are executed.

Control Block	When executed
FIRST PAGE HEADER	Before the program begins to process the first page
PAGE HEADER	Before the program begins to process the each subsequent page (and before the first page, if no FIRST PAGE HEADER clause is specified)
BEFORE GROUP OF	Before the program begins to process a group of stored records
ON EVERY ROW	When the program passes each record to the report
AFTER GROUP OF	After the program processes a group of stored records
PAGE TRAILER	After the program processes the end of each page
ON LAST ROW	After the program passes the last record to the report

The FORMAT control blocks may contain most of SQL and most of 4GL statements. The syntax of the statements used within FORMAT control blocks is the same as their syntax elsewhere in the program.

Statements in the Format Control Blocks

However, there is a list of statements that cannot appear within any of the FORMAT control blocks. If you try to include them, a compile-time error will appear:

- CONSTRUCT
- DEFER
- DEFINE
- DISPLAY ARRAY
- FUNCTION
- INPUT
- INPUT ARRAY
- MAIN
- MENU
- PROMPT
- REPORT
- RETURN

On the other hand, the FORMAT section allows several statements which cannot be used anywhere else in the program. One of them is the PRINT statement which plays an important part in the report output. This statement is used in a non-default report to specify which values and in which order should be printed to the report.



ON EVERY ROW Control Block

The ON EVERY ROW control block is used to specify the actions that 4GL must take for each new input record that was passed to the report definition. It is one of the most important clauses, as it processes the information passed to the report in the order it is passed and prints it to the report body. There are other clauses which can print to the report body (i.e. BEFORE GROUP OF, AFTER GROUP OF), but their output is conditioned by the record grouping.

The syntax of the ON EVERY ROW control block is:

```
ON EVERY ROW statement [, statement]
```

The ON EVERY ROW block is the block which specifies what data will be sent to the report and how it will be done. Therefore, the block must contain an output statement.

The output statement which sends the data stored in a report to the report output is the PRINT statement. The syntax and the effect of the PRINT statement operation are similar to those of the [DISPLAY](#) statement. The PRINT statement can include a set of operators which are already familiar to you (e.g., [COLUMN](#), [CLIPPED](#) operators, etc.). You cannot use the PRINT statement outside the REPORT control block.

Here is the simplest example of the PRINT statement specification within the ON EVERY ROW statement:

```
REPORT myrep(arg)  
  
DEFINE arg RECORD LIKE custom.*  
  
FORMAT  
ON EVERY ROW PRINT arg.*  
  
END REPORT
```

In this example, the program will pass the values stored to the program record *arg* to the report output each time the cursor moves to a new row. Thereafter, the record *arg* gets a new value which is also passed to the record output.

The FIRST PAGE HEADER Control Block

The FIRST PAGE HEADER control block is used to specify the actions that the program must take before it begins to process the first input record. For example, the control block can specify what will appear at the top of the first page of the report output, i.e. the name and the author of the report, etc.

The syntax of the control block is as follows:

```
FIRST PAGE HEADER statement [, statement...]
```

The FIRST PAGE HEADER control block is the control block which is processed before any output is generated. So, this control block can include the initialization of the variables that will be used by the FORMAT section in future:

```
...  
FORMAT  
FIRST PAGE HEADER  
LET cus_t = 1
```



```
LET av_r = sys_t/av_num
...
```

This control block won't be executed, if the program passes no data to the report, even if a report driver includes START REPORT and FINISH REPORT statements.

You can use the FIRST PAGE HEADER control block to specify a title page and column headings with the help of the PRINT statement. During the creation of the first page of the report output, this control block overrides a PAGE HEADING control block, if any.

```
...
FORMAT
  FIRST PAGE HEADER
    PRINT "This is the report No. ", rep_no
...
...
```

The TOP MARGIN option specified in the OUTPUT section or within the WITH clause of the STAT REPORT statement determines the free space that will appear between the page header and the top of the page.

The PAGE HEADER Control Block

The PAGE HEADER control block is used to specify the actions that the program must take before it starts processing each new page of the report.

The PAGE HEADER control block can display some information, column headings, page number, etc.

The syntax of the PAGE HEADER control block is as follows:

```
PAGE HEADER statement [, statement]
```

If the FORMAT section includes both FIRST PAGE HEADER and PAGE HEADING control blocks, the first page will have the heading specified in the former control block, and all the other pages will get the heading specified in the latter one.

The following example demonstrates how column names can be specified within the PAGE HEADER control block:

```
PAGE HEADER
  PRINT "ID",
  COLUMN 5, "Name", COLUMN 15, "Last name", COLUMN 25, "Age",
  COLUMN 30, "Hometown"
```

This control block will be executed each time a new page is created. At the top of the page a line will appear which contains column headings *ID*, *Name*, *Last name*, *Age*, *Hometown*.



The BEFORE GROUP OF Control Block

The BEFORE GROUP OF control block is used to specify the actions that the program must take before it processes the specified group of input records. The order of the group procession is determined by the ORDER BY option which appears within the SELECT statement or within the report definition.

The order of values sent to the report can also be specified in the ORDER BY clause of the REPORT program block, which will be discussed later in this chapter. If sort lists are specified in both report driver and report definition, the one in the REPORT program block is used by the program, and the other is ignored.

If no sorting is done and you still include the BEFORE GROUP OF clause, it is still executed each time the group variable changes value, but as no sorting is applied this may make your report quite messy.

The syntax of the BEFORE GROUP OF statement is:

```
BEFORE GROUP OF group_variable
    statement [, statement...]
```

The group variable here stands for the identifier of the variable included to the formal arguments list in the report definition. The variable specified in the BEFORE GROUP OF control block must be specified within the DEFINE section of the report definition.



Note: It is important that the name of the column (or variable) specified in the ORDER BY section of the report or in the ORDER BY clause of the SELECT statement matches the name of the group variable used in the AFTER GROUP OF control block.

A *group of records* is a set of input records that contain the same value for the variable specified after the BEFORE GROUP OF or AFTER GROUP OF keywords. When the BEFORE GROUP OF block is processed, the report variables get values from the first record of a new group. This clause is executed each time the value of the group variable changes.

The following snippet of a source code displays a message describing the group variable before grouping is performed:

```
DECLARE cur CURSOR FOR SELECT * FROM custom_list
OEDER BY hometown -- specifies the column by which the records are
sorted
...
REPORT my_first (rec_1)
...
FORMAT
    BEFORE GROUP OF hometown
    PRINT "Customers from ", hometown
        -- name of the variable (hometown) is the same as
        -- the name of the column the records are sorted by
...
END REPORT
```



Grouping by several Group Variables

The ORDER BY clause of a SELECT statement or the ORDER BY section of a report can reference more than one variable. However, the BEFORE GROUP OF block can reference only one group variable. To make the program perform some activities taking into consideration several groups of records, you need to specify several BEFORE GROUP OF clauses.

The variables references by the ORDER BY clause or the ORDER BY section have their order of precedence which influences the behaviour of the BEFORE GROUP OF clause. As has been already discussed for the ORDER BY clause, first the records are sorted by the first variable referenced (which is of the highest priority), then by the second and so on. Then the program executes each BEFORE GROUP OF block in the order from highest to lowest priority when begins to process the report.

If there is more than one BEFORE GROUP OF clause, each of them is executed:

- Whenever the value of the group variable changes
- Wherever the value of a variable with higher priority changes.

In the following example we have the SELECT statement which sorts the records by two variables and two BEFORE GROUP OF clauses which reference these variables in the FORMAT section of a report:

```
DECLARE my_cur FOR SELECT * FROM items ORDER BY item_type,
item_price
...
REPORT my_rep(items_rec)
DEFINE
    my_rep RECORD LIKE items.*
FORMAT
    BEFORE GROUP OF item_type
        PRINT "The list of ", item_type, " items"
    BEFORE GROUP OF item_price
        PRINT "With price ", item_price
    ON EVERY ROW PRINT items.*
...
END REPORT
```

As we declared a local record like a database table, the names of the variables match the names of the columns. The BEFORE GROUP OF *item_type* clause will be executed each time the value of *item_type* variable changes. Whereas the BEFORE GROUP OF *item_price* clause is executed each time the value of *item_type* and/or *item_price* variable changes, because the *item_price* variable has lower priority. The position of these clauses in the REPORT program block does not influence the order and peculiarities of their execution.

AFTER GROUP OF Control Block

You can use the AFTER GROUP OF control block to specify the actions that 4GL must perform after processing a group of sorted records. The grouping is performed according to the parameters specified in the ORDER BY section of the report definition or in the ORDER BY clause of a SELECT statement.

The syntax and the behaviour of the AFTER GROUP OF clause is the same as of the BEFORE GROUP OF clause. It also references a group variable. One group variable can have only one BEFORE GROUP OF and one AFTER GROUP OF clause. The syntax of the control block is as follows:

```
AFTER GROUP OF variable statement
```



The requirements for the variables specified within this block are similar to the requirements for the variables that follow the BEFORE GROUP OF statement. The variable specified in the AFTER GROUP OF control block must be specified within the DEFINE section of the report definition.

When the program executes the AFTER GROUP OF control block, it passes the values of the last record of the specified group to the local variables.

The AFTER GROUP OF control block is executed when:

- The value of the group variables changes;
- The value of a higher priority group variable changes;
- The program reaches the end of the report.

The order of precedence among several group variables specified within the sort list and the way the input records are sorted according to a group variable are described above for the BEFORE GROUP OF clause and are the same. The frequency of the group variable value changes depends partially on whether the SELECT statement has sorted the input records.

If the SELECT statement has already sorted the input records before they are passed to the report, the program executes the AFTER GROUP OF control blocks after the last record of the group is processed.

If the records haven't been sorted yet, the program can execute the AFTER GROUP OF statement after any record, because the value of the group variable may change after each record. If you don't include the ORDER BY section to the report definition, the program will execute the AFTER GROUP OF control blocks in the order in which they are listed in the FORMAT section.

If you specify only one variable in the BEFORE GROUP OF or AFTER GROUP OF control blocks, and the SELECT statement has already sorted the input records in sequence on that variable, you don't have to include the ORDER BY section to the report definition, if the records are sorted by the SELECT statement.

The PAGE TRAILER Control Block

The PAGE TRAILER control block is used to specify what information is to appear at the bottom of each output page.

The syntax of the block is as follows:

```
PAGE TRAILER statement [, statement]
```

If the report output needs a new page, the PAGE TRAILER control block is executed before the PAGE HEADER block.

The distance between the page trailer and the bottom of the page is defined by the BOTTOM MARGIN option.

The number of lines passed by the PAGE TRAILER control block to the bottom of the page cannot vary from page to page and must be accurately defined in the control block specification.

```
PAGE TRAILER
```



PRINT "See the next page"

The ON LAST ROW Control Block

The ON LAST ROW control block is used to specify the actions that are to be taken by the program when the last input record is passed to the report definition and the program encounters the FINISH REPORT statement.

The syntax of the block is as follows:

```
ON LAST ROW statement [, statement]
```

The statements that comprise the ON LAST ROW control block are executed after the ON EVERY ROW and AFTER GROUP OF blocks, if any.

When the ON LAST ROW block is executed, the variables processed by the report still have the values passed from the last report record.

You can add aggregate functions to the ON LAST ROW control block to display report totals. The report aggregate functions resemble the SQL aggregate functions and are described in the next chapter.

If the program encounters and executes the TERMINATE REPORT statement, the ON LAST ROW control block is skipped altogether.

The OUTPUT Section

The OUTPUT section duplicates the functionality of the TO and WITH clauses of the START REPORT statement. It is used to specify the format of the report output and its destination. The section may contain any elements valid for the [TO](#) and [WITH](#) clauses of the START REPORT statement. They can be specified in any order you want and their syntax is similar to that of the elements used within the START REPORT statement. Thus the OUTPUT section specifies the report destination and the dimensions of the report page.

```
OUTPUT
[ TOP | LEFT | BOTTOM | RIGHT MARGIN size ] | [ PAGE LENGTH size ]
[ REPORT TO
    SCREEN | PRINTER | FILE filename | PIPE program [ IN FORM | IN LINE
    MODE ] ]
```

Here *size* must be a literal integer specifying the number of lines or columns. The *filename* represents the file to which the report is to be printed and the *program* specifies the name of the program to which it is to be redirected. These issues are described in more details in the START REPORT statement.

	<p>Note: If there is a corresponding START REPORT statement specifying some of the options, the program will use its options and ignore the OUTPUT section of the report definition.</p>
--	---

You can also use the APPEND keyword to append the results of the current report to the results of the previous one, if it was performed to a file. As with the START REPORT statement, the APPEND keyword must



go before the TO FILE keywords. If the program does not find the file specified in the START REPORT...APPEND TO FILE statement, a run-time error will occur.

The following examples illustrate the usage of the OUTPUT section:

```
REPORT my_report (arg)
DEFINE arg RECORD LIKE custom.*  
  
OUTPUT
    REPORT TO FILE "my_report_file"--specifies the output file
    TOP MARGIN 2           -- sets the top margin size to 2
    BOTTOM MARGIN 2        -- sets the bottom margin to 2
    LEFT MARGIN 2          -- sets the left margin to 2
    PAGE LENGTH 25         -- sets the size length of the page to
                           25
--- OR ---  
  
OUTPUT
    REPORT APPEND TO FILE "my_report_file"--specifies the output
                           file
    TOP MARGIN 2           -- sets the top margin size to 2
    BOTTOM MARGIN 2         -- sets the bottom margin to 2
--- OR ---  
  
OUTPUT
    REPORT TO SCREEN -- specifies the Report window as the output
                           -- destination
--- OR ---  
  
OUTPUT
    LEFT MARGIN 5 -- sets the bottom margin to 5
    RIGHT MARGIN 125 -- sets the right margin to 25
--- OR ---  
  
OUTPUT
    TOP MARGIN 0 -- sets the top margin size to 0
    BOTTOM MARGIN 5 -- sets the bottom margin to 5
    REPORT TO "my_report_file" --specifies the output file
```

Remember, that any REPORT block can contain only one OUTPUT section, which can include any combination of the possible elements.

The ORDER BY Section

The ORDER BY section is used to specify the way the program should sort input records as well as to determine the sequence in which the GROUP OF control blocks in the FORMAT section are executed. This works as a substitute for the sorting done by the SELECT statement with the ORDER BY clause.



The syntax of the ORDER BY section is as follows:

```
ORDER [EXTERNAL] BY argument [ASC/DESC] [,argument [ASC/DESC]]
```

The *argument* stands for the name of the argument which must be specified in the report declaration (REPORT *report_name* (*argument*)). The program sorts the input records according to the values in the columns that correspond to the variables in the argument list. The list of variables specified here is called the *sorting list*.

If you don't include the ORDER BY section to the REPORT block, the input records will be passed to the report output in the order in which they are retrieved from the report driver.

The Sorting List

The order of the variables, specified in the ORDER BY section defines the order in which the records will be sorted by the program. If the sorting list includes several variables, 4GL sorts the records by these variables in left-to-right sequence. Thus if the ORDER BY clause contains two variables, the records are sorted by the first variable and then by the second one. The first variable has the highest priority. This method is the same as for the [ORDER BY](#) clause of a SELECT statement. You can sort only by the variables that appear in the argument list of the REPORT statement. The following snippet of a source code demonstrates how the *ord_price* values are sorted in ascending order:

```
REPORT orders_rep (rec, ord_price)  
  
DEFINE rec RECORD LIKE orders.*  
  
          Ord_price INTEGER  
  
ORDER BY ord_price
```

If you do not add the DESC keyword after the variable specified after the ORDER BY keywords, the records will be sorted in lowest-to-highest (or ascending) order by values of the highest-priority first variable. The ascending order can be explicitly set by the ASC keyword following a variable. The records that have the same value for the first variable will be ordered by values of the second variable, and so on.

If the records are sorted by a character data type variable, the program uses the code-set order by default.

The DESC keyword makes the program sort the records in highest-to-lowest order (the descending order). Precedence of the variables within the sort list is the same as that for the ascending sorting.

The sorting of the input records can also be performed by the ORDER BY clause of the SELECT statement specified within the report driver. If the sort lists are specified both in the SELECT statement and within the report definition, the one within the report definition has the higher precedence.

If all the input records are returned by a single cursor, and you want the program to process the report faster, specify the ORDER BY clause within the SELECT statement, rather than within the ORDER BY section of the report definition.



The sequence in which variables are listed in the sorting list influence the sequence of BEFORE GROUP OF and AFTER GROUP OF control blocks execution, regardless of the sequence in which the GROUP OF statements appear within the FORMAT section. For example, if the variables are listed in the ORDER BY section as *var1*, *var2*, *var3*, the program will process the control blocks in the following order:

BEFORE GROUP OF var1	--1
BEFORE GROUP OF var2	--2
BEFORE GROUP OF var3	--3
ON EVERY ROW	--4
AFTER GROUP OF var3	--3
AFTER GROUP OF var2	--2
AFTER GROUP OF var1	--1

The EXTERNAL Keyword

If you have already used the SELECT statement to sort the input records, use the ORDER EXTERNAL BY statement to control the order in which the GROUP BY control blocks are executed.

If the ORDER BY section of the report definition does not include the EXTERNAL keyword or the report definition specifies aggregate functions operating with all the input records, the program passes twice through the input records. When it passes for the first time, it uses the database server to sort the data and then creates a temporary file to save the sorted values. During the second pass, the program calculates the aggregate values and produces output using the data stored in the temporary files.

When the EXTERNAL keyword is present, the program needs to make only one pass through the data: it doesn't need to create a temporary table for sorting the input records. The EXTERNAL keyword doesn't let the program sort the data twice and can result in better performance of the report.

Example

This application creates a number of reports. It is required that you are connected to the database specified in the DATABASE statement (you can change the database name specified in the DATABASE statement to match one of your existing databases). You must also run the example application from the [Databases in 4GL](#) chapter before running this example to create tables in the database. It is not necessary to populate these tables, because this example uses temporary tables, but the regular tables must exist in the database in order for the DEFINE LIKE statements to function correctly.

```
#####
# This application produces formatted reports with various
# page parameters and with grouping of the content.
#####
DATABASE db_examples

DEFINE
    # The structure of the temporary table and the regular
    # clients table are the same so this saves code, because
    # we cannot define variables like a temporary table
    r_clients      RECORD LIKE clients.*,
    error_code,
    num_key        INTEGER

MAIN
```



```
# Function prep_data_for_report is located in a function library
# it creates temporary tables and populates them with data
# to make the report more visual
CALL prep_data_for_report()

# One of the cursors we declare does not use any sorting
# It will be used to send data to reports
DECLARE c_clients_1 CURSOR FOR
    SELECT clients_temp.* FROM clients_temp
    WHERE id_client > 10

# The other cursor declared includes sorting of values
# it will also be used to send data to reports
DECLARE c_clients_2 CURSOR FOR
    SELECT clients_temp.* FROM clients_temp
    WHERE id_client > 10
    ORDER BY clients_temp.id_country, clients_temp.city, clients_temp.gender,
            clients_temp.first_name

MENU "Report"
    COMMAND "Default Margins"
        "Formatting a report with the default page margins"
        # Starting a report with the default page margins.
        START REPORT cli_list_1 TO SCREEN
            -- if you comment the TO SCREEN keywords
            -- the report will be sent to
            -- clients_list_1.rep file,
            -- because the OUTPUT section is present
            -- in the report block.
            FOREACH c_clients_1 INTO r_clients.*
                OUTPUT TO REPORT cli_list_1(r_clients.*)
            END FOREACH
        FINISH REPORT cli_list_1

    COMMAND "Non_Default Margins"
        "A report with non-default page margins"
        # Creating a report with the page margins specified in the OUTPUT
        # section of the REPORT block.
        START REPORT cli_list_2 TO SCREEN
            FOREACH c_clients_2 INTO r_clients.*
                OUTPUT TO REPORT cli_list_2(r_clients.*)
            END FOREACH
        FINISH REPORT cli_list_2

    COMMAND "Headers and Trailers"
        "A report with page headers and trailers in the FORMAT section"
        # This report is formatted due to control clauses in the FORAMT section
        # of the REPORT program block declaring report cli_list_3.
        START REPORT cli_list_3 TO SCREEN
            FOREACH c_clients_2 INTO r_clients.*
                OUTPUT TO REPORT cli_list_3(r_clients.*)
            END FOREACH
        FINISH REPORT cli_list_3
```



```
COMMAND "Complicated Report"
    "A report with all the control blocks available"
    # This report uses all the possible formatting clauses
    # available in the FORMAT section
    START REPORT cli_list_4 TO SCREEN
        FOREACH c_clients_2 INTO r_clients.*
            OUTPUT TO REPORT cli_list_4(r_clients.*)
        END FOREACH
    FINISH REPORT cli_list_4

COMMAND "Report Termination"
    "The report is terminated before it is complete"
    # If an error occurs, variable error_code will have a non-zero value
    # and the execution of report will be terminated by
    # the TERMINATE REPORT. In this case some data will be lost, as
    # the AFTER GROUP OF,PAGE TRAILER and ON LAST ROW will not be
    # executed after the last line is sent to the report.
    START REPORT cli_list_4 TO SCREEN
        LET error_code = 0           -- reset the variable to 0
        FOREACH c_clients_2 INTO r_clients.*
            IF r_clients.id_client = 50
            THEN
                LET r_clients.id_country = "000" -- we purposefully assign
                                            -- an invalid value
            END IF
            OUTPUT TO REPORT cli_list_4(r_clients.*)
            IF r_clients.id_country = "000"
            THEN
                LET error_code = error_code + 1 -- then we set the variable
                                                -- tracking the errors
                                                -- to a non-null value
            END IF
        END FOREACH

        IF error_code > 0 -- then we specify what actions should be taken,
                           -- if an error occurred
        THEN TERMINATE REPORT cli_list_4 -- we terminate the report,
                                         -- if there is an error
        ELSE FINISH REPORT cli_list_4   -- we finish it normally,
                                         -- if there is no error
        END IF

    COMMAND "Exit"
        EXIT MENU

    END MENU

END MAIN

#####
# This simple report section prints the contents of r_clients record
# to report pages with the default dimensions. The report records are
# ordered in the report. This report is a two pass one.
#####
```



```
REPORT cli_list_1(r_clients)
  DEFINE
    r_clients RECORD LIKE clients.*

  OUTPUT
    REPORT TO "clients_list_1.rep" -- this line is directing
                                    -- the output to a file
                                    -- but it has no effect, because it is
                                    -- overridden
                                    -- by the START REPORT statement,
                                    -- unless you
                                    -- comment the TO SCREEN keywords there

  # The order in which the values are printed does not depend on
  # the SELECT statement which retrieved these values. It depends on their
  # order in the ORDER BY statement
  # Thus this will be a two pass report - first values are retrieved
  # by the SELECT statement and stored in a temporary table, and during the
  # second pass they are ordered.
  ORDER BY r_clients.id_country, r_clients.city, r_clients.gender,
           r_clients.first_name

  FORMAT
    ON EVERY ROW
      PRINT r_clients.id_client USING "&&&"," ",r_clients.id_country,
        " ",r_clients.first_name, r_clients.last_name,r_clients.gender,
        " ",r_clients.city,r_clients.address[1,33], r_clients.start_date,
        " ",r_clients.end_date

  END REPORT
#####
# This report block prints the values to the report pages with the margins
# specified in the OUTPUT section.
#####
REPORT cli_list_2(r_clients)
  DEFINE
    r_clients RECORD LIKE clients.*

  OUTPUT
    TOP      MARGIN 0
    LEFT     MARGIN 0
    BOTTOM   MARGIN 10 -- 10 lines from the bottom will remain empty
    PAGE LENGTH 30   -- this means that the report page is 30 lines long
                      -- the width of the page remains default

  # The records of the report are ordered by the SELECT statement
  # declared for c_clients_2 cursor, which has the ORDER BY clause.
  # In this case it is a one pass report which is more effective.
  ORDER EXTERNAL BY   r_clients.id_country, r_clients.city, r_clients.gender,
                    r_clients.first_name

  FORMAT
    ON EVERY ROW
      PRINT r_clients.id_client USING "&&&"," ",r_clients.id_country,
        " ",r_clients.first_name, r_clients.last_name,r_clients.gender,
```



```
" ",r_clients.city,r_clients.address[1,33], r_clients.start_date,  
" ",r_clients.end_date  
  
END REPORT  
  
#####
# This REPORT program block prints the contents of r_clients record.  
# The report is formatted using the FORMAT control blocks, the records  
# are ordered in the report driver.  
#####  
REPORT cli_list_3(r_clients)  
  DEFINE  
    r_clients RECORD LIKE clients.*,  
    var_char CHAR(120)  
  
  # You can comment any of the control clauses below, recompile  
  # and run the program  
  # to see which control block is responsible for this or that part  
  # of the formatting  
  FORMAT  
    FIRST PAGE HEADER  
      LET var_char = "This text is displayed by the <FIRST PAGE HEADER>" ,  
          " clause at the top of the first page"  
      PRINT var_char  
  
    PAGE HEADER  
      LET var_char = "This text is displayed by the <PAGE HEADER> clause" ,  
          " at the top of every page except the first one"  
      PRINT var_char  
  
    ON EVERY ROW -- this clause prints the information to the report body  
                  -- each time new values are passed to record r_clients  
      PRINT r_clients.id_client USING "&&&"," ",r_clients.id_country,  
        " ",r_clients.first_name, r_clients.last_name,r_clients.gender,  
        " ",r_clients.city,r_clients.address[1,33],r_clients.start_date,  
        " ",r_clients.end_date  
  
    PAGE TRAILER  
      LET var_char = "This text is displayed by the <PAGE TRAILER> clause" ,  
          " at the bottom of each report page"  
      PRINT var_char  
  
    ON LAST ROW  
      LET var_char = "This text is displayed by the <ON LAST ROW> caluse" ,  
          "after the last record is passed and printed"  
      PRINT var_char  
  
END REPORT  
#####
# This REPORT program block prints the contents of r_clients. The report  
# is formatted using all the possible control blocks of the FORMAT section  
# The report records are ordered using the FORMAT clause of the block.  
#####
REPORT cli_list_4(r_clients)
```



```
DEFINE
    r_clients RECORD LIKE clients.*,
    var_char CHAR(120)

# You can comment any of the control clauses below, recompile
# and run the program
# to see which control block is responsible for this or that part
# of the formatting
FORMAT
    FIRST PAGE HEADER
        LET var_char = "This text is displayed by the <FIRST PAGE HEADER> ,
                      " clause at the top of the first page"
        PRINT var_char

    PAGE HEADER
        LET var_char = "This text is displayed by the <PAGE HEADER> clause",
                      " at the top of every page except the first one"
        PRINT var_char

# The BEFORE GROUP OF clauses can be used only if there is some
# ordering applied.
# In this case the ordering is done outside the report with the help
# of the SELECT statement with the ORDER BY clause for which the cursor
# retrieving the report data is declared.
BEFORE GROUP OF r_clients.id_country
    LET var_char = "This text is displayed by the <BEFORE GROUP OF> ,
                  " clause before the country code value changes"
    PRINT var_char

BEFORE GROUP OF r_clients.city
    LET var_char = "This text is displayed by the <BEFORE GROUP OF> ,
                  " clause before the city value changes"
    PRINT var_char

ON EVERY ROW
    PRINT r_clients.id_client USING "&&&," ,r_clients.id_country,
                                " ",r_clients.first_name, r_clients.last_name,r_clients.gender,
                                " ",r_clients.city,r_clients.address[1,33],r_clients.start_date,
                                " ",r_clients.end_date

# The AFTER GROUP OF clauses can also be used only if some
# ordering applied either by the SELECT statement with the ORDER BY
# clause, or by the ORDER BY section of the REPORT program block
AFTER GROUP OF r_clients.city
    LET var_char = "This text is displayed by the <AFTER GROUP OF> clause",
                  "after the city value changes"
    PRINT var_char

AFTER GROUP OF r_clients.id_country
    LET var_char = "This text is displayed by the <AFTER GROUP OF> clause",
                  "after the country code value changes"
    PRINT var_char

PAGE TRAILER
    LET var_char = "This text is displayed by the <PAGE TRAILER> clause",
```



```
" at the bottom of each report page"
PRINT var_char

ON LAST ROW
LET var_char = "This text is displayed by the <ON LAST ROW> clause",
           "after the last record is passed and printed"
PRINT var_char

END REPORT
```

The Function Library

This function library populates the temporary tables with different sorts of data to create enough various records to constitute illustrative reports and enable sorting and ordering of values in them. If this function is not executed, the reports will be empty, even if they are processed successfully.

```
#####
# This function is used to add some data to the database table
# in order to make the reports more illustrative when the data are grouped
# and sorted by the different report control blocks.
#####
DATABASE db_examples

FUNCTION prep_data_for_report()
DEFINE
    # the structure of regular and temporary tables are the same
    # so these records, though defined like regularly tables,
    # can be used with temporary tables
    r_clients    RECORD LIKE clients.*,
    r_contracts  RECORD LIKE contracts.*,
    i            INTEGER

    # We create temporary tables with the same structure
    # as regular tables to prevent possible errors while
    # adding values to the tables due to constraint violation.
    # It also prevents us from distorting the information
    # stored in the regular tables and saves the efforts
    # for deleting the inserted information
CREATE TEMP TABLE clients_temp
(
    id_client      SERIAL,
    id_country     CHAR(3),
    first_name     CHAR(20),
    last_name      CHAR(20),
    date_birth     DATE,
    gender         CHAR(6),
    id_passport    CHAR(10),
    date_issue     DATE,
    date_expire    DATE,
    address        CHAR(80),
    city           CHAR(20),
```



```

        zip_code      CHAR(6),
        add_data      CHAR(512),
        start_date    DATE,
        end_date      DATE
    )
CREATE TEMP TABLE contracts_temp
(
    id_contract   SERIAL,
    id_type       INTEGER,
    id_client     INTEGER,
    id_currency   CHAR(3),
    num_con       CHAR(10),
    account       CHAR(14),
    amount         DECIMAL(16,2),
    interest_rate DECIMAL(6,2),
    start_date    DATE,
    end_date      DATE
)

BEGIN WORK

LET i = 1 -- we set the counter to 10, it will be used as the id
          -- and for other purposes while assigning values

# Then we add new clients who live in Great Britain, Germany,
# the USA and Ukraine.
# The clients added are living in different cities, which will help
# to make the ordering and grouping more visual.
# We also add a deposit contract for each of them.

WHILE TRUE
    INITIALIZE r_clients.* ,r_contracts.* TO NULL
    LET i = i + 1
    IF i > 210 THEN
        EXIT WHILE -- this loop will be repeated, until
                    -- 200 records are inserted
    END IF

    # The complex system of the IF statements below allows us
    # to insert the values we need depending on the client id
    IF i >= 11 AND i <= 65
    THEN # We add clients living in Germany for rows from 11 to 65
        LET r_clients.id_country  = "276"
        IF i >= 11 AND i <= 25           -- some of them living in Munich
        THEN LET r_clients.city = "Munich"-- we add the city information
            IF i >= 11 AND i <= 20
                # And other information required
                THEN LET r_clients.gender = "male"
                    LET r_contracts.id_type      = 1
                    LET r_contracts.id_currency = "978"
                    LET r_contracts.amount     = 15000.00
                    LET r_contracts.interest_rate = 12.50
                ELSE LET r_clients.gender = "female"
                    LET r_contracts.id_type      = 1
                    LET r_contracts.id_currency = "978"
    END IF
)

```



```
        LET r_contracts.amount          = 7000.00
        LET r_contracts.interest_rate = 11.00
    END IF
END IF
IF i > 25 AND i <= 37 -- the others living in Berlin
THEN LET r_clients.city = "Berlin"
    IF i >= 25 AND i <= 32
        THEN LET r_clients.gender = "male"
            LET r_contracts.id_type      = 2
            LET r_contracts.id_currency = "978"
            LET r_contracts.amount      = 25000.00
            LET r_contracts.interest_rate = 15.70
        ELSE LET r_clients.gender = "female"
            LET r_contracts.id_type      = 2
            LET r_contracts.id_currency = "978"
            LET r_contracts.amount      = 22000.00
            LET r_contracts.interest_rate = 17.00
        END IF
    END IF
ELSE LET r_clients.city = "Bonn"
    IF i >= 37 AND i <= 51
        THEN LET r_clients.gender = "male"
            LET r_contracts.id_type      = 1
            LET r_contracts.id_currency = "840"
            LET r_contracts.amount      = 8000.00
            LET r_contracts.interest_rate = 18.30
        ELSE LET r_clients.gender = "female"
            LET r_contracts.id_type      = 1
            LET r_contracts.id_currency = "840"
            LET r_contracts.amount      = 3000.00
            LET r_contracts.interest_rate = 16.40
        END IF
    END IF
ELSE LET r_clients.city = "Dresden"
    IF i >= 57 AND i <= 59
        THEN LET r_clients.gender = "male"
            LET r_contracts.id_type      = 3
            LET r_contracts.id_currency = "978"
            LET r_contracts.amount      = 18000.00
            LET r_contracts.interest_rate = 12.70
        ELSE LET r_clients.gender = "female"
            LET r_contracts.id_type      = 3
            LET r_contracts.id_currency = "978"
            LET r_contracts.amount      = 13000.00
            LET r_contracts.interest_rate = 11.00
        END IF
    END IF
END IF
IF i > 65 AND i <= 125
THEN # We add clients living in Great Britain for rows from 66 to 125
    LET r_clients.id_country = "826"
    IF i >= 65 AND i <= 78
        THEN LET r_clients.city = "London"
```



```
IF i >= 65 AND i <= 78
THEN LET r_clients.gender = "male"
    LET r_contracts.id_type      = 1
    LET r_contracts.id_currency  = "978"
    LET r_contracts.amount       = 50000.00
    LET r_contracts.interest_rate = 15.80
ELSE LET r_clients.gender = "female"
    LET r_contracts.id_type      = 1
    LET r_contracts.id_currency  = "978"
    LET r_contracts.amount       = 45500.00
    LET r_contracts.interest_rate = 14.70
END IF
END IF
IF i > 78 AND i <= 88
THEN LET r_clients.city = "Liverpool"
    IF i >= 78 AND i <= 87
    THEN LET r_clients.gender = "male"
        LET r_contracts.id_type      = 2
        LET r_contracts.id_currency  = "826"
        LET r_contracts.amount       = 150000.00
        LET r_contracts.interest_rate = 16.50
    ELSE LET r_clients.gender = "female"
        LET r_contracts.id_type      = 2
        LET r_contracts.id_currency  = "826"
        LET r_contracts.amount       = 130500.00
        LET r_contracts.interest_rate = 12.90
    END IF
END IF
IF i > 88 AND i <= 91
THEN LET r_clients.city = "Manchester"
    IF i >= 88 AND i <= 91
    THEN LET r_clients.gender = "male"
        LET r_contracts.id_type      = 3
        LET r_contracts.id_currency  = "826"
        LET r_contracts.amount       = 780000.00
        LET r_contracts.interest_rate = 19.00
    ELSE LET r_clients.gender = "female"
        LET r_contracts.id_type      = 3
        LET r_contracts.id_currency  = "826"
        LET r_contracts.amount       = 650800.00
        LET r_contracts.interest_rate = 18.10
    END IF
END IF
IF i > 91 AND i <= 107
THEN LET r_clients.city = "Southampton"
    IF i >= 91 AND i <= 105
    THEN LET r_clients.gender = "male"
        LET r_contracts.id_type      = 1
        LET r_contracts.id_currency  = "840"
        LET r_contracts.amount       = 2800000.00
        LET r_contracts.interest_rate = 14.20
    ELSE LET r_clients.gender = "female"
        LET r_contracts.id_type      = 1
        LET r_contracts.id_currency  = "840"
        LET r_contracts.amount       = 17500000.00
```



```
        LET r_contracts.interest_rate = 13.70
    END IF
END IF
IF i > 107 AND i <= 125
THEN LET r_clients.city = "Canterbury"
    IF i >= 107 AND i <= 119
    THEN LET r_clients.gender = "male"
        LET r_contracts.id_type      = 2
        LET r_contracts.id_currency = "826"
        LET r_contracts.amount      = 930000.00
        LET r_contracts.interest_rate = 15.10
    ELSE LET r_clients.gender = "female"
        LET r_contracts.id_type      = 2
        LET r_contracts.id_currency = "826"
        LET r_contracts.amount      = 750000.00
        LET r_contracts.interest_rate = 14.20
    END IF
END IF
END IF
IF i > 125 AND i <= 160
THEN # We add clients living in Ukraine for rows from 126 to 160
    LET r_clients.id_country = "804"
    IF i >= 125 AND i <= 135
    THEN LET r_clients.city = "Kyiv"
        IF i >= 125 AND i <= 133
        THEN LET r_clients.gender = "male"
            LET r_contracts.id_type      = 1
            LET r_contracts.id_currency = "980"
            LET r_contracts.amount      = 50000.00
            LET r_contracts.interest_rate = 22.10
        ELSE LET r_clients.gender = "female"
            LET r_contracts.id_type      = 1
            LET r_contracts.id_currency = "980"
            LET r_contracts.amount      = 350000.00
            LET r_contracts.interest_rate = 21.30
        END IF
    END IF
    IF i > 135 AND i <= 144
    THEN LET r_clients.city = "Kharkiv"
        IF i >= 135 AND i <= 141
        THEN LET r_clients.gender = "male"
            LET r_contracts.id_type      = 2
            LET r_contracts.id_currency = "980"
            LET r_contracts.amount      = 123350000.00
            LET r_contracts.interest_rate = 19.70
        ELSE LET r_clients.gender = "female"
            LET r_contracts.id_type      = 2
            LET r_contracts.id_currency = "980"
            LET r_contracts.amount      = 20000000.00
            LET r_contracts.interest_rate = 28.00
        END IF
    END IF
    IF i > 144 AND i <= 155
    THEN LET r_clients.city = "Lviv"
        IF i >= 144 AND i <= 150
```



```
THEN LET r_clients.gender = "male"
    LET r_contracts.id_type      = 1
    LET r_contracts.id_currency  = "980"
    LET r_contracts.amount       = 5000.00
    LET r_contracts.interest_rate = 15.00
ELSE LET r_clients.gender = "female"
    LET r_contracts.id_type      = 1
    LET r_contracts.id_currency  = "980"
    LET r_contracts.amount       = 3000.00
    LET r_contracts.interest_rate = 13.40
END IF
END IF
IF i > 155 AND i <= 160
THEN LET r_clients.city = "Odesa"
    IF i >= 155 AND i <= 158
        THEN LET r_clients.gender = "male"
            LET r_contracts.id_type      = 2
            LET r_contracts.id_currency  = "840"
            LET r_contracts.amount       = 5550000.00
            LET r_contracts.interest_rate = 15.80
        ELSE LET r_clients.gender = "female"
            LET r_contracts.id_type      = 2
            LET r_contracts.id_currency  = "840"
            LET r_contracts.amount       = 1234000.00
            LET r_contracts.interest_rate = 15.00
        END IF
    END IF
END IF
IF i > 160
THEN # We add clients living in the USA for all rows from 161
    LET r_clients.id_country  = "840"
    IF i >= 160 AND i <= 193
        THEN LET r_clients.city = "New York"
            IF i >= 160 AND i <= 185
                THEN LET r_clients.gender = "male"
                    LET r_contracts.id_type      = 1
                    LET r_contracts.id_currency  = "840"
                    LET r_contracts.amount       = 80000000.00
                    LET r_contracts.interest_rate = 12.00
                ELSE LET r_clients.gender = "female"
                    LET r_contracts.id_type      = 1
                    LET r_contracts.id_currency  = "840"
                    LET r_contracts.amount       = 40000000.00
                    LET r_contracts.interest_rate = 11.80
                END IF
            END IF
        END IF
        IF i > 193 AND i <= 203
            THEN LET r_clients.city = "Chicago"
                IF i >= 193 AND i <= 200
                    THEN LET r_clients.gender = "male"
                        LET r_contracts.id_type      = 2
                        LET r_contracts.id_currency  = "840"
                        LET r_contracts.amount       = 5780000.00
                        LET r_contracts.interest_rate = 25.20
                    ELSE LET r_clients.gender = "female"
```



```

        LET r_contracts.id_type      = 2
        LET r_contracts.id_currency = "840"
        LET r_contracts.amount     = 8360000.00
        LET r_contracts.interest_rate = 23.70
    END IF
END IF
IF i > 203 AND i <= 210
THEN LET r_clients.city = "Detroit"
    IF i >= 203 AND i <= 208
    THEN LET r_clients.gender = "male"
        LET r_contracts.id_type      = 3
        LET r_contracts.id_currency = "840"
        LET r_contracts.amount     = 123456000.00
        LET r_contracts.interest_rate = 16.60
    ELSE LET r_clients.gender = "female"
        LET r_contracts.id_type      = 3
        LET r_contracts.id_currency = "840"
        LET r_contracts.amount     = 104578000.00
        LET r_contracts.interest_rate = 18.50
    END IF
END IF
END IF
# Then we assign the rest of the data which will not be used
# for sorting and therefore can be identical for all the records
LET r_clients.id_client   = i -- we assign the counter as the client id
LET r_clients.first_name  = "Fname ",i USING "<<<<&"
LET r_clients.last_name   = "Lname ",i USING "<<<<&"
LET r_clients.id_passport = "Passp.",i USING "&&&&"
LET r_clients.address    = "New client's address ",i USING "<<<<&"
LET r_clients.start_date  = TODAY
LET r_clients.end_date    = DATE("12/31/9999")

# We insert the values assigned earlier into temporary table
INSERT INTO clients_temp VALUES(r_clients.*)

# We also create a deposite contract for each created client
LET r_contracts.id_contract = 0
LET r_contracts.id_client   = r_clients.id_client
LET r_contracts.num_con     = i USING "&&&&"
IF r_contracts.id_type = 1
THEN LET r_contracts.account = "263030900",r_clients.id_client USING
    "&&&"
ELSE IF r_contracts.id_type = 2
    THEN LET r_contracts.account = "263530900", r_clients.id_client
        USING "&&&"
    ELSE LET r_contracts.account = "262030900", r_clients.id_client
        USING "&&&"
    END IF
END IF
LET r_contracts.start_date = TODAY
LET r_contracts.end_date   = DATE("12/31/9999")
# We insert values into a temporary table
INSERT INTO contracts_temp VALUES(r_contracts.*)

END WHILE

```



COMMIT WORK

END FUNCTION



Statements Used in Report Section

In the previous chapter, we have discussed the structure of the REPORT program block and the ways it is activated and executed. We found out that the REPORT program block is a complicated one and is bound by more strict rules than the MAIN and the FUNCTION program blocks.

One of the significant differences of the REPORT program block is the fact that it can include the statements unavailable in other program blocks. These statements are called *report execution statements*. They can appear only within the FORMAT section of the REPORT program block, and are generally used to control the report workflow.

There are five report execution statements:

Statement	Effect
PRINT	Adds a specified item to the report output.
EXIT REPORT	Terminates the report processing
NEED	Initiates the page break unless the current page of the report includes the specified line number.
PAUSE	Allows the user to control the scrolling of the record output, if it is performed to the screen.
SKIP	Adds blank lines into a report or breaks the page.

This chapter is dedicated to the detailed description of these statements.

The PRINT Statement

The PRINT statement is the most essential statement of the report definition. It is the PRINT statement that produces the report output from the report definition.

We have briefly discussed the PRINT statement in the previous chapter, and now it's time to go into detail.

The PRINT statement can be followed by one or more items from the list given below, if one PRINT statement includes more than one clause, they should be separated by commas:

- A 4GL expression or a variable.
- COLUMN operator.
- SPACE operator.
- PAGENO operator - available only in the PRINT statement.
- LINENO operator - available only in the PRINT statement.
- BYTE *variable* operator.
- TEXT *variable* operator.
- FILE "*filename*".
- Aggregate report function.
- Character expression.

Remember, that the program cannot display the BYTE values; it just outputs references to them. Any variable the value of which is sent to the output by the PRINT statement must be of module or global scope or be declared within the DEFINE section of the report definition.



The Output Character Position

The output produced by the PRINT statement begins at the current position of the cursor, unless the output character position is changed by some operators. The default current position for each PRINT statement is the first character position on the line. Thus each PRINT statement prints information beginning with a new line. Margin and header specifications can also influence the default position for the first character.

If you do not use the CLIPPED or USING operator, the program displays values one by one with widths specified by the declared data types and data types sizes.

By default, each PRINT statement produces one line of the output, unless it includes the WORDWRAP option or displays a file. The following example creates three lines of the output:

```
PRINT "Customer: ", id
PRINT f_name, l_name, age
PRINT hometown
```

You can add a semicolon to the end of the output list following the PRINT keyword. This semicolon means that the output from the next PRINT statement will be displayed to the previous line. The following snippet of a source code also prints the output on three lines:

```
PRINT "Customer: ", id
PRINT f_name, l_name;
PRINT age
PRINT hometown
```

The FILE, TEXT, and BYTE Keywords

The FILE option is used to specify the file from which a multiple-line character string is to be inserted to the output. The syntax of the option is:

```
PRINT FILE filename
```

The *filename* stands for the value name of a text file whose contents you want to include into the report.

The TEXT keyword must be followed by a variable of the TEXT data type whose value the PRINT statement will send to the output:

```
PRINT TEXT variable
```

The BYTE keyword must be followed by the variable of the BYTE data type. However, you cannot really send a BYTE value to a report, the <byte value> string will be printed instead of the byte value.

```
PRINT BYTE variable
```



Note: If the PRINT statement has either TEXT or FILE keywords, you cannot specify any other clauses for additional output in the same PRINT statement.



Expressions and Operators Used with the PRINT Statement

The expression list following the PRINT keyword must return one or more values that the program can display as printable characters. The table below lists the expressions that can follow the PRINT keyword. Some of these expressions can appear only within the FORMAT section of REPORT program block; these statements are marked with asterisks. Some of the statements can be used only in the FORMAT section of the report definition or within SQL statements; these are marked with apostrophes.

ASCII	DATE()	MIN() '	SUM() '
AVG() '	DAY()	MDY()	TIME
CLIPPED	EXTEND()	MONTH()	TODAY
COLUMN	GROUP() *	ORD()	UNITS
COUNT '	LENGTH()	PAGENO() *	USING
CURRENT	LINENO *	PERCENTO *	WEEKDAY()
DATE	MAX() '	SPACES *	YEAR()

If a DATE or MONEY value used within the expression list is formatted with the USING operator, the format string specified within the USING operator overrides the values stored in the DBDATE, DBFORMAT, and DBMONEY environment variables.

The functions and operators specific to the PRINT statement are described in the sections below

Operators Used in the PRINT Statement

The PRINT statement allows two operators to be used to specify blank spaces between the values of one report record. These are the COLUMN operator which we can also use for the DISPLAY statement, and the SPACE operator, which is valid only in the PRINT statement.

There are some operators specific to the PRINT statement which deals with the report pages and report lines. You can also use such operators as ASCII, ORD(), and others listed in the table above.

The ASCII Operator

The ASCII operator is used to return the character whose numeric code is specified after the ASCII keyword. The performance of the [ASCII](#) operator has already be described in the previous chapters, but it has one peculiarity when used in the PRINT statement within the report definition. If you want the PRINT statement to add a NULL character to the report, specify 0 following the ASCII operator in the PRINT statement. For example, the following line will make the program print the NULL character:

```
PRINT ASCII 0
```

You should remember, that 0 in the ASCII operator makes it print NULL only within the PRINT statement. In other statements (i.e. in DISPLAY statement), it will produce a blank space.

The COLUMN Operator

You can use the COLUMN operator within the PRINT statement when you want to move the character position forward within the current line. Its function is the same as it has for the DISPLAY statement. The COLUMN operator has the next syntax:

```
COLUMN integer_expression
```



The integer expression must return a positive integer value specifying the character position offset from the left margin. This value must be equal to or less than the line width, i.e., then the difference between the right margin and the left margin. The value that must be printed form the column specified in the COLUMN operator must follow the integer expression and be separated from it with a comma:

```
PRINT "REPORT FORM", COLUMN 80, DATE
```

The COLUMN operator specifies an absolute character position on a report line which does not depend on the current cursor position. However, if the current position is greater than the value returned by the *integer expression*, the program ignores the COLUMN operator.

The SPACE Operator

The SPACE or SPACES operator is used to add a string of blank spaces which is equal to a quoted string containing the specified number of blank spaces (ASCII 32). Unlike the COLUMN operator, it specifies a relative position on a report line which depends on the current cursor position and is greater than the current position by the number of columns specified as the operand.

The syntax of the SPACE or SPACES operator is as follows:

```
integer_expression SPACE | SPACES
```

The integer expression must return a positive integer determining the offset from the current character position which is no greater than the difference between the current position and the right margin.

When the SPACE or SPACES operator is used within the PRINT statement, the blank spaces are inserted into the current position. When the operator is executed, it moves to the right the new current character position, and it changes by the specified number of characters.

The following example specifies the distance between the columns within the report output:

```
PRINT id, 5 SPACES, f_name, 2 SPACES, l_name, 2 SPACES, age
```

When the PRINT statement sends the data to the output, the distance between the value of the variable *id* will be 5 blank spaces, all the other columns will be separated by two blanks.

The SPACE Operator Outside the PRINT Statement

The SPACE (or SPACES) operator can be used outside the PRINT statement:

- In a DISPLAY statement it inserts the specified number of blank spaces into the displayed value:

```
DISPLAY var1, 3 SPACE, var2 AT 2,2
```

- In a LET statement it can be used to assign blank spaces to a character variable as its value. In this case the SPACE operator and its operand must be included into parentheses:

```
LET a = (10 SPACES)
LET b = (6 SPACES), "=ZIP" -- concatenates a value with white
spaces
```



The LINENO Operator

This operator needs no operand. It returns the value which corresponds to the line number of the current report line. The program gets the line number by calculating the number of the lines from the top of the page, including the lines in the top margin.

In the following example, the program is forced to print the line numbers at the beginning of the line. The first four lines of the page are given to the top margin, so, the data output begins only from the line 5:

```
IF (LINENO>4) THEN  
  
    PRINT COLUMN 2, LINENO, 2 SPACE, custom_data.*  
  
END IF
```

You can add the USING clause to the LINENO operator in order to set a pattern according to which the line number will be displayed. You can add a character string to the value displayed by the LINENO operator and set the format of the value by means of the USING operator. The details of the [USING](#) operator performance are described in previous chapters.

```
IF (LINENO>4) THEN  
  
    PRINT COLUMN 2, LINENO USING "LINE <<<"  
  
END IF
```

The PAGENO Operator

The PAGENO operator is similar to the LINENO operator, but it returns the number of the currently printed page. It also needs no operand.

You can create an output like "PAGE x OF y" where x is the current page and y is the total number of pages (e.g. Page 5 of 32). To do this, you have to use the [COUNT\(*\)](#) aggregate within the SELECT statement in order to find the number of rows that will be sent to the report output. The number of the rows that appear on each page must also be known and fixed.

In the following example, the program calculates the total number of pages and adds a report page number:

```
DATABASE my_test_db  
MAIN  
  
DEFINE pers RECORD  
    id INT,  
    f_name,l_name CHAR (20),  
    age INT  
END RECORD,  
  
line_num INT -- a variable to store the number of lines  
  
SELECT COUNT(*) id INTO line_num FROM custom --returns the number of lines  
--in the table  
DECLARE cur CURSOR FOR SELECT * FROM custom
```



```
START REPORT myrep

    FOREACH cur INTO pers./*
        #sends the input records and the total number of lines to the report
        OUTPUT TO REPORT myrep(pers.* , line_num)

    END FOREACH

    FINISH REPORT myrep

END MAIN

-----
REPORT myrep(a,line_num)

DEFINE a RECORD LIKE custom.* ,
    num, line_num INTEGER

OUTPUT
    PAGE LENGTH 30 -- specifies the page length
    TOP MARGIN 0
    BOTTOM MARGIN 0

FORMAT
    FIRST PAGE HEADER
    LET num = line_num/30+1      --counts the total number of pages. +1 is
                                -- necessary if the result isn't likely to
                                -- be a whole number. If this is the case,
                                -- the last page will be ignored
    PAGE HEADER

    PRINT PAGENO USING "PAGE <<& , " of " , num --prints PAGE x OF y
    ON EVERY ROW PRINT a.* --outputs the input records

END REPORT
```

The WORDWRAP Operator

If a line outputted by the report is longer than the page width, you can use the WORDWRAP operator to wrap subsequent segments of long character strings onto subsequent lines of report output. The string values which are too long and don't fit the space between the current position and the right margin are divided into segments, and the program creates temporary margins to display them. In this case, the current character position becomes the left margin; the default right margin is 132 or that specified in the RIGHT MARGIN clause of the OUTPUT section. The temporary margins override the default ones or those specified in the OUTPUT section.

The syntax of the WORDWRAP statement is the following:



```
[Character expression / TEXT variable] WORDWRAP [RIGHT MARGIN temporary]
```

Here “temporary” stands for a literal integer that specifies the character position of a temporary right margin. The example below will print the value of the “story” variable with each line consisting of 65 symbols:

```
PRINT COLUMN 5, story WORDWRAP RIGHT MARGIN 70
```

Splitting Data with WORDWRAP Operator

The string of the data outputted to the report can include different ASCII characters, as well as the TAB (ASCII 9), LINEFEED (ASCII 10), and ENTER (ASCII 13) characters. These characters divide the output string into words, comprised by substrings of other printable characters. If the outputted line is too long to fit the screen line, the program breaks the outputted line at the words division and adds blank spaces to the end of the line. When breaking the line, 4GL keeps to the following rules (they are given in descending order of precedence):

- Break any LINEFEED, ENTER, or LINEFEED, ENTER pair.
- Break at the last TAB character of blank space (ASCII 32) before the right margin.
- Break at the right margin, unless character farther to left is a blank, ENTER, LINEFEED, or TAB character.

When the PRINT statement prints the TAB character, the program expands it with blank spaces until it reaches the next tab stop. By default, the tab stops are located in every eighth column counting from the left edge of the page.

4GL keeps to the page discipline when the WORDWRAP option is specified. If the current string is too long, 4GL executes the statements specified in page header and page trailer blocks before it continues output to a new page.

The specifics of the WORDWRAP operator performance may vary depending on the locales.

Aggregate Report Functions

The *aggregate report functions* are used to summarize data stored in several records of a report. The syntax and the effects of these aggregate functions are similar, but not equal to those of SQL aggregates. For the aggregate functions of the SQL statements see [“Advanced Cursor Usage” chapter](#).

The generalized syntax of the aggregate function usage within the PRINT statement is as follows:

```
PRINT [GROUP] aggregate [WHERE expression]
```

The *aggregate* stands for an aggregate function, which may be PERCENT(), COUNT(), AVG(), SUM(), MIN(), or MAX() function. These functions cannot be used within an expression, this means that each aggregate function must be separated from the other parts of a PRINT statement and cannot be combined with any operator.

Aggregate Dependencies

The aggregative value can depend on a group of records (then it is specified by the GROUP keyword) or on a condition (specified by the WHERE clause). If an aggregate function does not depend on any of the above mentioned, it depends on all the records sent to the report.



The GROUP Keyword

The GROUP keyword is optional and can be included only within an AFTER GROUP OF control block. This keyword is not valid in any other report clause. It makes the aggregate function use data only for a *group* of records that have the same value for a variable specified in the AFTER GROUP OF control block. Otherwise the aggregate functions will perform calculations using all the values sent to the report.

The WHERE Clause

The WHERE clause is similar to the WHERE clause of the SELECT statement and allows you to select and operate with the records that satisfy the Boolean condition specified in this clause.

An Aggregate Depending on all the Records

If an aggregate has no GROUP keyword and no WHERE clause, it depends on all the records. If this aggregate appears anywhere except the ON LAST ROW clause, the argument of the aggregate function must be included into the report arguments.

Calculating Average and Total Values

The AVG() and SUM() aggregates are used to evaluate the arithmetic mean value and the total, respectively, of *expression* among all the records that are outputted to the report, or among records, selected by the GROUP or WHERE clauses. The name of the report argument for which you want to receive the average or total value must be placed in the parentheses.

```
REPORT my_rep(col1, col2, clo3)
      DEFINE col1, col2, clo3 VARCHAR (30)
      FORMAT
      ...
      AFTER GROUP OF col1
          PRINT "The total sum of ", col1, " for this group
is "
          GROUP SUM(col1)
      ON LAST ROW
          PRINT "The average value of ", col1, AVG(col1)
          PRINT "The average value of ", col1,
          "in cases when ", col2, " is M: ",
          AVG(col1) WHERE col2 MATCHES "M"
```

The example above will calculate the total sum for each group and the average value of the same variable depending on all the records and on the records which have "M" for *col2* value.

	Note: These aggregate functions work only with numeric values.
--	---



Calculating the Number of Records

The COUNT(*) and PERCENT(*) aggregates are used to return, respectively, the total number of the records chosen by the WHERE clause, and the percentage of the overall number of records in the current report. You must include the asterisk (*) symbol into the parentheses following the aggregate names.

The COUNT(*) function returns the total number of records passed to the record, if the GROUP keyword and WHERE clause are absent. If the GROUP clause is present, it returns the number of records in a group, if the WHERE clause is present, it returns the number of records which meet the WHERE condition.

The PERCENT(*) function returns the percentage of the records which meet certain condition:

- It returns "100", if there is no WHERE clause, which means that 100% of report records correspond to the requirement.

```
PRINT PERCENT(*)
```

- If there is the WHERE clause, 4GL calculates the percent of records which meet the condition of the WHERE clause.

```
PRINT PERCENT(*) WHERE age>20
```

- If there is both GROUP keyword and WHERE clause, 4GL calculates the percentage of record meeting the condition in the group of records (Remember that the GROUP keyword can appear only if the function is used in the AFTER GROUP OF clause).

```
PRINT GROUP PERCENT(*) WHERE age>20
```

- If there is only the GROUP keyword, it returns "100", because without a condition all the records of the group are taken into account and all records make 100%.

Calculating Maximum and Minimum Values

The MIN() and MAX() aggregates return, respectively, the minimum and the maximum values of the number, currency, and INTERVAL values specified as its argument among all the records that have been passed to the report output or among the records selected by the GROUP keyword or the WHERE clause.

For DATE and DATETIME values, *maximum* means the latest, and *minimum* means the earliest point in time.

Character strings are evaluated according to their first character. In default English locales, *maximum* means after and *minimum* means before in the ASCII collating sequence, where 1<A<a. If the first characters are equal, the values are sorted by the second characters, etc.

If you use the aggregate functions within the AFTER GROUP OF control block, you can use the GROUP keyword to qualify them:

```
AFTER GROUP OF orders
      PRINT "The average price is ", GROUP MAX(price)
```

The GROUP keyword can appear only within the AFTER GROUP OF control block.



EXIT REPORT Statement

You can use the EXIT REPORT statement to terminate the processing of the report and to return the program control to the report driver statement that follows the most recently executed OUTPUT TO RECORD statement.

The syntax of the statement is very simple:

```
EXIT REPORT
```

When the program executes the EXIT REPORT statement, it terminates the processing of the current report and deletes all the temporary and intermediate items that were created during the processing of this report.

In fact, the effect of the EXIT REPORT statement is similar to that of the TERMINATE REPORT statement; the difference is that the EXIT REPORT statement can appear only within the corresponding report definition, and the TERMINATE REPORT keywords can be used only within the corresponding report driver. The TERMINATE REPORT statement used within a report definition can appear only in a nested report driver invoking another report.

The NEED Statement

The NEED statement makes any subsequent display start from the new page if the distance between the current line and the bottom margin is less than it is specified in the statement.

The syntax of the NEED statement is:

```
NEED lines LINES
```

Lines here stands for an integer expression returning a positive whole number. This number must be less than the page length.

You can use the NEED statement in order not to let the program divide the parts of the output that you want to be outputted together. The following example demonstrates the NEED statement in action:

```
NEED 8 LINES
PRINT cust.*
```

If the current page has fewer than 8 lines, the program will add the PAGE TRAILER to it and create a new page with a PAGE HEADER.

When the NEED statement evaluates the number of the lines remaining on the page, it does not take the BOTTOM MARGIN into account. The NEED statement cannot be included into the FIRST PAGE HEADER, PAGE HEADER, or PAGE TRAILER blocks.

The PAUSE Statement

You can use the PAUSE statement to suspend the output temporary until the user resumes it by pressing the RETURN key.

The syntax of the PAUSE statement is as follows:



```
PAUSE ["string"]
```

The *string* stands for an optional quoted string displayed when the PAUSE statement is executed. If you don't specify any string, no message will be displayed.

The PAUSE statement performs differently in Windows and in Linux/UNIX systems.

The PAUSE Statement on Linux/UNIX

If the program encounters the PAUSE statement, the output remains on the screen until the user presses the RETURN key.

The PAUSE statement will have no effect, if the START REPORT statement includes the TO clause with other than the SCREEN keyword. It will also have no effect on the output, if the OUTPUT section contains the REPORT TO keywords and they are not followed by the SCREEN keyword.

The PAUSE statement is usually specified in the PAGE HEADER or PAGE TRAILER clause. In the following example, it is the PAGE TRAILER clause which contains the PAUSE statement, which means, that the new page of the report output won't be displayed until the user presses the RETURN key.

```
PAGE TRAILER
PAUSE "Press RETURN to view the next page of the report output"
```

The PAUSE Statement on Windows

Querix has designed the Report Viewer, a tool which is a part of graphical clients and which is used to display the report output sent to the screen. The Report Viewer has already described in the previous chapter.

When the program performs a report, it sends all the result to the Report Viewer. The user can view any page of the report at any time using the arrow keys on the Viewer toolbar.

When the application is run in character mode, the whole report is sent to the Notepad and is opened automatically.

In both these cases the user has the full control over the report view and can see any part of the report whenever they want. Therefore, there is no use in the PAUSE statement and the program ignores it.

The SKIP Statement

The SKIP statement is used to insert blank lines to the report output or to move the next character to the top of the next page of the report output.

The syntax of the SKIP statement can be of two types:

- SKIP TO TOP OF PAGE
- SKIP *integer* LINE

The *integer* stands for a literal integer specifying the number of lines to be skipped. You can use LINE or LINES keyword after this integer.



The TOP OF THE PAGE option makes the program add a page trailer to the current page and moves the current character position to the first line of the next page (if the TOP MARGIN is specified, the *first line* means the first line after the top margin)

Here is an example of the SKIP statement application:

```
FIRST PAGE HEADER
    PRINT "CUSTOMER LIST"

    SKIP 1 LINE -- 1 line is inserted between the two outputted
rows

    PRINT "Customers from ", cus_city
```

The following example makes each group of records be displayed to a different page:

```
FORMAT

    ON EVERY ROW PRINT x./*

    AFTER GROUP OF x.id SKIP TO TOP OF PAGE
```

Example

The following example uses the aggregate functions and special operators together with the PRINT statement to make the report nicely formatted and easy to understand.

To run this application, you need to be connected to the database specified in the DATABASE statement. The database must contain the tables created previously by the example in [Databases in 4GL chapter](#). They should be populated using the [Populating Database Tables](#) example. This example also references the function library used for the previous example, so you need to add this library to the program requirements to be able to run this application.

```
#####
# This is the example of reports with different formatting options
# as well as with different aggregate functions applied.
#####
#DATABASE db_examples

MAIN
DEFINE
    r_clients      RECORD LIKE clients.*,
    r_contracts    RECORD LIKE contracts.*,
    r_currency     RECORD LIKE currency.*,
    r_country_list RECORD LIKE country_list.*,
    num_key        INTEGER

    # Function prep_data_for_report is located in a function library
    # it creates temporary tables and adds the data to them
    # to make the report more visual
    CALL prep_data_for_report()
```



```
# We declare a number of cursors with different ordering options
# for future use. We use both regular and temporary tables for the
# queries. The temporary tables are used where we need a variety
# of data to sort. These are the cleints_temp and contracts_temp
# tables created by the function above
DECLARE c_contracts_1 CURSOR FOR
    SELECT contracts_temp.* FROM contracts_temp
    WHERE contracts_temp.id_contract > 6
    ORDER BY contracts_temp.id_type, contracts_temp.id_currency

DECLARE c_contracts_2 CURSOR FOR
    SELECT contracts_temp.* FROM contracts_temp
    WHERE contracts_temp.id_contract > 6 AND
        contracts_temp.id_type IN(1,3)
    ORDER BY contracts_temp.id_type, contracts_temp.id_currency

DECLARE c_clients_1 CURSOR FOR
    SELECT clients_temp.*,country_list.*
    FROM clients_temp,country_list
    WHERE clients_temp.id_country = country_list.id_country AND
        clients_temp.id_client > 10
    ORDER BY clients_temp.id_country, clients_temp.city, clients_temp.gender

DECLARE c_clients_and_contracts CURSOR FOR
    SELECT clients_temp.*,contracts_temp.*,currency.*
    FROM clients_temp,contracts_temp,currency
    WHERE contracts_temp.id_currency = currency.id_currency AND
        clients_temp.id_client = contracts_temp.id_client AND
        clients_temp.id_client > 10
    ORDER BY contracts_temp.id_currency, contracts_temp.amount DESC,
        clients_temp.id_client

MENU "Printing"

    COMMAND "PRINT Operators"
        "Produces a report using the PRINT statement with operators"

        # This report driver invokes the report block
        # which PRINT statements contain special operators.
        START REPORT contracts_list_1 TO SCREEN
        FOREACH c_contracts_1 INTO r_contracts.*
            OUTPUT TO REPORT contracts_list_1(r_contracts.*)
        END FOREACH
        FINISH REPORT contracts_list_1

    COMMAND "WORDWRAP and PRINT"
        "Formatting a report using the WORDWRAP together with the PRINT"

        START REPORT contracts_list_2 TO SCREEN
        FOREACH c_contracts_2 INTO r_contracts.*
            OUTPUT TO REPORT contracts_list_2(r_contracts.*)
        END FOREACH
        FINISH REPORT contracts_list_2
```



```
COMMAND "Headings"
        "Formatting a report to create a comprehensible heading"

        START REPORT clients_list_1 TO SCREEN
        FOREACH c_clients_1 INTO r_clients.* ,r_country_list.*
                OUTPUT TO REPORT clients_list_1(r_clients.* )
        END FOREACH
        FINISH REPORT clients_list_1

COMMAND "COUNT(*) and PERCENT(*)"
        "Formatting a report using functions COUNT(*) and PERCENT(*)"

        START REPORT clients_list_2 TO SCREEN
        FOREACH c_clients_1 INTO r_clients.* ,r_country_list.*
                OUTPUT TO REPORT clients_list_2(r_clients.* ,r_country_list.* )
        END FOREACH
        FINISH REPORT clients_list_2

COMMAND "Aggregate Functions"
        "Formatting a report using functions SUM( ),AVG( ),MAX( ), and MIN( )"

        START REPORT contracts_list_3 TO SCREEN
        FOREACH c_clients_and_contracts
                INTO r_clients.* ,r_contracts.* ,r_currency.* 
                OUTPUT TO REPORT contracts_list_3(r_clients.* ,
                        r_contracts.* ,r_currency.* )
        END FOREACH
        FINISH REPORT contracts_list_3

COMMAND "Exit"
        EXIT PROGRAM

END MENU

END MAIN
#####
# This REPORT block prints the contents of r_contracts
# using different operators available in the PRINT statement.
#####
REPORT contracts_list_1(r_contracts)
    DEFINE
        r_contracts RECORD LIKE contracts.* ,
        var_char      CHAR(120)

    OUTPUT
        LEFT   MARGIN 0
        BOTTOM MARGIN 0

    FORMAT
        # We use the page header to display page numbers and the headers
        PAGE HEADER
            PRINT COLUMN 80,"Page ",PAGENO USING "<<<" -- we print the number of
            -- the current page
            -- at the top of each page
        SKIP 1 LINE
```



```

LET var_char = "lineno type cur num account amount percent",
        " start_date end_date"
PRINT COLUMN 3,var_char
SKIP 1 LINES
ON EVERY ROW
    PRINT COLUMN 6,LINENO USING "<<&"; -- the next PRINT statement will
        -- print to the same line
        -- due to the semicolon

    PRINT COLUMN 11, r_contracts.id_type USING "&,3 SPACE",
        r_contracts.id_currency, 2 SPACES,
        r_contracts.num_con[1,4]; -- the next PRINT statement
        -- will print to the same line
PRINT COLUMN 26,r_contracts.account,r_contracts.amount USING "-,-,-,-,-&.&&",
        r_contracts.interest_rate,COLUMN 66,
        r_contracts.start_date, COLUMN 78,r_contracts.end_date
ON LAST ROW
SKIP 1 LINES
# The ASCII can also be used in the PRINT statement
# The "ASCII operator" string is encoded here
PRINT COLUMN 6,ASCII 84,ASCII 104,ASCII 101,ASCII 32,ASCII 65,
        ASCII 83,ASCII 67,ASCII 73,ASCII 73, ASCII 32,
        ASCII 111,ASCII 112,ASCII 101,ASCII 114,ASCII 97,
        ASCII 116,ASCII 111,ASCII 114,ASCII 13

END REPORT

#####
# This report block prints the contents of the report using different PRINT
# operators including the WORDWRAP operator.
#####
REPORT contracts_list_2(r_contracts)
DEFINE
    r_contracts RECORD LIKE contracts.*,
    var_char CHAR(120),
    v_name_type LIKE type_contract.name_type

OUTPUT
    LEFT MARGIN 0
    TOP MARGIN 0
    BOTTOM MARGIN 0

FORMAT
    FIRST PAGE HEADER
        LET var_char = "id_type name_type cur num account",
            " amount percent start_date end_date"
        PRINT COLUMN 2,var_char
        SKIP 1 LINES
    # We populate the file which we will then print,
    # each time r_contracts.id_type value changes
    # The contents of the file will be rewritten each time
    # the contract type changes
    BEFORE GROUP OF r_contracts.id_type
        SELECT t.name_type INTO v_name_type FROM type_contract t
        WHERE t.id_type = r_contracts.id_type

```



```

ON EVERY ROW
    PRINT COLUMN 6,r_contracts.id_type USING "<& ",COLUMN 10,
# we print the file using the WORDWRAP, the file contains the names
# of the contract types; you will see how the names will be wordwrapped
# and the rest of the data printed by this PRINT statement will be
# printed on the line where the file data ends
        v_name_type CLIPPED WORDWRAP RIGHT MARGIN 21,
        COLUMN 24, r_contracts.id_currency,
        2 SPACES,r_contracts.num_con[1,4], COLUMN 35,
        r_contracts.account,r_contracts.amount USING "-,-,-,-,-&.&&",
        r_contracts.interest_rate,COLUMN 75,
        r_contracts.start_date,COLUMN 87,r_contracts.end_date
ON LAST ROW
SKIP 2 LINES
PRINT COLUMN 6,"Beginning with this place the contents of",
                " file_to_print.txt will be added to the report"
SKIP 2 LINES
PRINT FILE "file_to_print.txt" -- this file can be any text file with
                                -- this name added to the source folder
                                -- and to the program requirements
END REPORT

#####
# This report block prints the client data. The data are ordered in the
# report driver. The formatting is done in the FORMAT section.
#####
REPORT clients_list_1(r_clients)
DEFINE
    r_clients RECORD LIKE clients.*,
    var_char CHAR(120),
    i_col     INTEGER

OUTPUT
    LEFT    MARGIN 0
    BOTTOM MARGIN 0

FORMAT
    FIRST PAGE HEADER
        LET var_char = "The list of Clients"
        LET i_col = 55 - LENGTH(var_char)/2
        PRINT COLUMN i_col,var_char CLIPPED
        SKIP 1 LINE
        PRINT COLUMN 2,"Run on ",TODAY USING "mm-dd-yyyy";
        PRINT COLUMN 102,"Page ",PAGENO USING "<<<"
        PRINT "-----",
        "-----"
        PRINT " Client| Client first and last names | Passport |",
                "          Client address      |Registration| Registration"
        PRINT "   ID  |                                | ID      |",
                "          date      |cancel. date "
        PRINT "-----",
        "-----"
    PAGE HEADER
        PRINT COLUMN 2,"Run on ",TODAY USING "mm-dd-yyyy";
        PRINT COLUMN 102,"Page ",PAGENO USING "<<<"

```



```

PRINT "-----",
"-----"
PRINT " Client| Client first and last names | Passport |",
"       Client address      |Registration| Registration"
PRINT "   ID |                                | ID |",
"           | date     |cancel. date "
PRINT "-----",
"-----"
ON EVERY ROW
PRINT COLUMN 3, r_clients.id_client USING "&&&",
COLUMN 9, r_clients.first_name,r_clients.last_name CLIPPED,
COLUMN 45,r_clients.id_passport,
COLUMN 58,r_clients.address[1,24] CLIPPED,
COLUMN 85,r_clients.start_date,
COLUMN 98,r_clients.end_date
PAGE TRAILER
PRINT "-----",
"-----"

END REPORT
#####
# This report block prints the client data grouped by countries and by cities.
# The report formatting as well calculations using COUNT(*) and PERCENT(*)
# functions are done in the FORMAT section.
#####
REPORT clients_list_2(r_clients,r_country_list)
DEFINE
    r_clients      RECORD LIKE clients.*,
    r_country_list RECORD LIKE country_list.*,
    v_percent_m,
    v_percent_f    DECIMAL(6,2),
    var_char        CHAR(120),
    i_col          INTEGER

OUTPUT
LEFT    MARGIN 0
TOP     MARGIN 0
BOTTOM MARGIN 0

FORMAT
FIRST PAGE HEADER
LET var_char = "The List of Clients Sorted by Countries and by Cities"
LET i_col = 55 - LENGTH(var_char)/2 -- here we calculate the coordinate
-- where to display the value
SKIP 2 LINE
PRINT COLUMN i_col,var_char CLIPPED
SKIP 1 LINE
PRINT COLUMN 2,"Run on ",TODAY USING "mm-dd-yyyy"
# We divide the groups using the BEFORE GROUP OF clauses
BEFORE GROUP OF r_clients.id_country
LET i_col = 55 - LENGTH(r_country_list.country_name)/2
SKIP 1 LINE
PRINT COLUMN i_col,UPSHIFT(r_country_list.country_name)
BEFORE GROUP OF r_clients.city
LET i_col = 55 - LENGTH(r_clients.city)/2

```



```

SKIP 1 LINE
PRINT COLUMN i_col,r_clients.city
# Before each group we print the header so that it was more convenient
# for the user
PRINT "-----",
"-----"
PRINT " Client| Client first and last names | Passport |",
"         Client address      |Registration| Registration"
PRINT "   ID |                                | ID |",
"           | date     |cancel. date "
PRINT "-----",
"-----"
ON EVERY ROW
PRINT COLUMN 3, r_clients.id_client USING "&&&",
COLUMN 9, r_clients.first_name,r_clients.last_name CLIPPED,
COLUMN 45,r_clients.id_passport,
COLUMN 58,r_clients.address[1,24] CLIPPED,
COLUMN 85,r_clients.start_date,
COLUMN 98,r_clients.end_date
AFTER GROUP OF r_clients.city
PRINT "-----",
"-----"
PRINT COLUMN 2,"Clients total number by City: ",GROUP COUNT(*)
USING "<<<&"
# This PRINT statement calculates the number of records for males and
# for females; the GROUP keyword before COUNT(*) is allowed only in the
# BEFORE GROUP OF and AFTER GROUP OF clauses. This keyword makes the
# function calculate only values included into the group.
PRINT COLUMN 2,"Including: ",GROUP COUNT(*)
WHERE r_clients.gender = "male" USING "<<<&," males, ",
GROUP COUNT(*) WHERE r_clients.gender = "female" USING "<<<&",
"females"
PRINT "-----",
"-----"
AFTER GROUP OF r_clients.id_country
PRINT COLUMN 2,"Clients total number by Country: ",GROUP COUNT(*)
USING "<<<&"
PRINT COLUMN 2, "Including: ",GROUP COUNT(*) WHERE
r_clients.gender = "male" USING "<<<&," males, ",
GROUP COUNT(*) WHERE r_clients.gender = "female"
USING "<<<&," females"
PRINT "-----",
"-----"
ON LAST ROW
LET v_percent_m = PERCENT(*) WHERE r_clients.gender = "male"
LET v_percent_f = PERCENT(*) WHERE r_clients.gender = "female"
PRINT COLUMN 2,"Clients total number: ",COUNT(*) USING "<<<&"
PRINT COLUMN 2,"Including: ",COUNT(*) WHERE r_clients.gender = "male"
USING "<<<&," males, ", COUNT(*) WHERE r_clients.gender = "female"
USING "<<<&," females (",v_percent_m USING "<<<&.&&","% males, ",
v_percent_f USING "<<<&.&&","% females)"
PRINT "=====",
"=====",
"====="

```



END REPORT

```

#####
# This report block prints the client data grouped by countries and cities.
# The report formatting as well calculations using SUM(),AVG(),MAX() and MIN()
# functions are done in the FORMAT section.
#####
REPORT contracts_list_3(r_clients,r_contracts,r_currency)
DEFINE
    r_clients      RECORD LIKE clients.*,
    r_contracts    RECORD LIKE contracts.*,
    r_currency     RECORD LIKE currency.*,
    var_char       CHAR(120),
    i_col          INTEGER

OUTPUT
    LEFT   MARGIN 0
    TOP    MARGIN 0
    BOTTOM MARGIN 0

FORMAT
    FIRST PAGE HEADER
        LET var_char = "Deposit contracts sorted by the deposit amount",
            " and interest rates"
        LET i_col = 48 - LENGTH(var_char)/2
        SKIP 2 LINE
        PRINT COLUMN i_col,var_char CLIPPED
        SKIP 1 LINE
        PRINT COLUMN 2,"Run on ",TODAY USING "mm-dd-yyyy"
    BEFORE GROUP OF r_currency.id_currency
        LET i_col = 48 - LENGTH(r_currency.code_currency)/2
        SKIP 1 LINE
        PRINT COLUMN i_col,r_currency.code_currency
        PRINT -----
        "-----"
        PRINT " Client| Client first and last names | Passport |",
              " Client address |Registration|Registration "
        PRINT " ID | | ID |",
              " date |cancel. date "
        PRINT -----
        "-----"
    ON EVERY ROW
        PRINT COLUMN 3, r_clients.id_client USING "&&&",
                  COLUMN 9, r_clients.first_name,r_clients.last_name CLIPPED,
                  COLUMN 42,r_contracts.amount USING "--,---,---,&.&&",
                  COLUMN 62,r_contracts.interest_rate USING "---&.&&",
                  COLUMN 72,r_contracts.start_date,
                  COLUMN 85,r_contracts.end_date
    AFTER GROUP OF r_currency.id_currency
        PRINT -----
        "-----"
        PRINT COLUMN 2,"Total amount\avg interest rate: ",COLUMN 42,
        # We calculate the total value of all the contracts in the group
        GROUP SUM(r_contracts.amount)USING "--,---,---,&.&&",
        COLUMN 62,
```



```
# Here we calculate the average sum of an interest rate in the group
GROUP AVG(r_contracts.interest_rate) USING "---&.&&"  

PRINT COLUMN 2,"Maximum amount\\interest rate: ",COLUMN 42,  

# We calculate the maximum sum of the contract in the group
GROUP MAX(r_contracts.amount)USING "--,---,---,--&.&&",  

COLUMN 62,  

# And the maximum interest rate in the group
GROUP MAX(r_contracts.interest_rate) USING "---&.&&"  

PRINT COLUMN 2,"Minimum amount\\interest rate: ",COLUMN 42,  

# This calculates the minimum contract sum
GROUP MIN(r_contracts.amount)USING "--,---,---,--&.&&",  

COLUMN 62,  

# and the minimum interest rate
GROUP MIN(r_contracts.interest_rate) USING "---&.&&"  

PRINT "=====,  

"=====,  

"====="  

  
END REPORT
```

The Function Library

This application utilizes the same library as [Formatting Report Output](#) example. You just need to add it to the program requirements without changing its source code. It creates some of the tables used for reporting and populates them with values.



Error Handling

Efficient database usage and management need proper execution of all the logical blocks of statements that are used to modify database structure and content. The situation, where only part of necessary statements are executed, and some are skipped, or when the program operation is terminated because of an error may cause confusions, invalid values, and, as the result - real problems for the user. That is why it is very important to be sure that each statement has been executed correctly and to handle the exceptions in time and properly.

The Notion of Errors and Warnings

The discussion of exceptions should start from the exceptions classifications and the reasons of their occurrence.

We can divide all the possible exceptions into two groups, which are errors and warnings.

An error is an exception that results in invalid values, statements execution failures, and other situations that, if not handled, make the normal program execution and the program task performance impossible and cause the program termination or prevent the program from being launched in the first place.

A warning is a situation, when the program detects one or several troubles during the execution or compilation, but these troubles are not fatal. However, they can result in errors or invalid values in future. When a warning occurs, the program is not terminated, but the problem is fixed in some of the built-in variables.

All the errors and warnings can be classified into compile-time and run-time ones.

- Compile-time errors and warnings. The 4GL language has its own systematic and restricted syntax which makes the language usage possible. When the language syntax is violated, the compiler cannot "understand" what the program is supposed to do, and produces an error during the compilation. Such error is called a compile-time error. A compile-time error can also occur if SQL syntax is violated or the environment variables have improper settings.

If the syntax is correct, but the database server or the compiler finds a potential problem, a warning is generated. The program will be compiled successfully, but it is advisable that you check everything in the source code in order to prevent run-time errors and invalid values.

- Runtime errors and warnings. Runtime errors occur when the syntax of the source code is correct, but the program or its part cannot be executed. This may result from invalid values, functions application, wrong delimiters, incompatible parameters, etc.

Runtime warnings occur when the program tries to perform an action, which can cause invalid program operations, but is not fatal for the program execution.



Runtime errors and warnings, in their turn, can be classified into three subgroups:

- o SQL errors, warnings, and end-of-data conditions that are diagnosed by the database server. These errors are recorded to the built-in record sqlca.
- o Interrupt, Quit, or other signals sent by the user or any other source. These are recorder into built-in variables int_flag and quit_flag correspondingly.
- o Runtime errors and warnings produced by the 4GL.

If you don't take any special measures, a runtime error will terminate the program execution immediately and will display a message describing the error. A runtime warning will not appear on the screen but will store the information about the problem to a built-in variable.

Each of these exceptions can be handled successfully by different means. You can control the processing of SQL errors and warnings, end-of-data conditions, errors in screen input-output operations and 4GL expressions evaluation by means of the WHENEVER statement.

You can use the [DEFER](#) statement to prevent the program from termination when the user presses the Quit or the Interrupt key.

You can handle the errors that occur during the SQL statements execution by referring to the built-in record sqlca.

All the methods which have not been touched upon earlier in this manual will be discussed below in this chapter.

The Status Built-In Variable

4GL uses the built-in variable status to record errors which occur during both SQL and 4GL statements execution. Status is a built-in variable of INTEGER data type, which gets a new value each time the program performs an operation (i.e. executes a statement). The value got by the variable status indicates the success of the operation performance.

If an operation is performed successfully, the program assigns 0 to the variable status. If an exception occurs during an operation performance, the operation cannot be performed due to any reason, or it results into invalid values, the variable status gets a negative value. This value is a code number of the exception that has occurred. In most cases, if there is no WHENEVER statement prior to the place of the error occurrence, the status variable will not get the number of the error, if this error does not prevent the program from execution (such as a conversion error).

The variable status is of the global scope and gets a new value irrespective of the program block or module where the executed statement is located.

The value of the variable status always indicates the success of the most recent statement execution. You should remember it, when you refer to the variable in the program:

```
...
IF a>100 THEN
    LET sal = 200
    IF status = 0 THEN ... END IF --the status indicates the success
                                -- of the execution of the LET
```



```
-- statement  
END IF  
IF status = 0 THEN... --the status indicates the success of  
--the execution of the IF statement
```



Note: The actual result of Boolean expressions evaluation (TRUE, FALSE) does not affect the value of the variable status. In other words, if the result is FALSE, the variable will still get the value 0, because the conditional statement which uses this expression is executed without errors.

The WHENEVER Statement

The WHENEVER statement is used to specify the actions to be taken by the program when warnings, errors, and absence of data occur during the runtime. The general syntax of the WHENEVER statement is:

```
WHENEVER event action
```

The *event* stands for an exceptional condition which can be trapped using the WHENEVER statement. The list of events includes ERROR, SQLERROR, WARNING, NOT FOUND, UNDEFINED FUNCTION and others. You will find their description later in this chapter.

The *action* denotes an action to be performed if the specified event takes place. The WHENEVER statement can replace a number of statements which determine the action to be implemented if an exceptional condition is satisfied after SQL statements and some of the 4GL statements execution. The actions can be represented by the STOP, CONINUE, CALL and GOTO statements. The STOP statement causes the program to abort once the specified event takes place. The CONTINUE statement, on the contrary, prevents the program from terminating whenever the specified event occurs. With the CALL statement specified, an event handler is invoked to fix the problem. The GOTO statement passes the program control to a labelled place in the code. All these statements are described below.

The ERROR Event

The ERROR event may trap only SQL errors, screen interaction statement errors, validation errors that occur during the execution of the VALIDATE statement, but ignore the errors generated while processing other 4GL expressions:

```
WHENEVER ERROR CONTINUE
```

In addition to the exceptions mentioned above, the ERROR event can handle errors produced by the program during the 4GL statements execution. Then the keyword ANY must be added before the word ERROR keyword. For example, the line of code below will prevent the program termination on discovering both SQL and 4GL errors.

```
WHENEVER ANY ERROR CONTINUE
```



Built-In Variables that Track Errors

When 4GL encounters the WHENEVER statement followed by the ERROR keyword, it takes the specified action depending on the value of the *sqlcode* member of the *sqlca* built-in variable of the RECORD data type. As it is very important in the context of error handling, we should say a few words about this variable and its member.

The built-in record *sqlca* is used to save the information about the result of the actions performance when the program is working with a database. The record includes five members, one of which is the integer built-in variable *sqlcode*. The program assigns a value to this variable each time an SQL statement is executed. Value 0 means that the SQL operation has been performed successfully. For example, if a statement is intended to return some data to a variable, the zero value will be assigned to the variable *sqlcode*, if the data is returned and can be assigned to the specified variable. A nonzero positive value means that the statement was executed, but didn't return any data that could be used. A negative value of the *sqlcode* variable means, that something has gone wrong during the statement execution. The negative number represents a code of the problem that has occurred.

Trapping Errors

The ERROR event used in the WHENEVER statement means that the following action is to be performed each time the *sqlcode* variable gets a negative value. The action specified is also taken when a VALIDATE statement or a screen interaction statement fails.

The following statement stops the program execution if an error occurs:

```
WHENEVER ERROR STOP
```

You can also pass the program control to some place in the program code by means of the GOTO statement, invoke a function which deals with the problem using the CALL statement, or make the program ignore the error by using the CONTINUE keyword.

The WHENEVER ERROR CONTINUE structure may prove very useful, if you want to refer to the variable status in your program and take actions that depend on the type of the error that has occurred:

```
...
WHENEVER ERROR CONTINUE
LABEL ent_col:
PROMPT "Enter the column name" FOR col_name
CALL take_values(col_name)
IF status=-217 THEN
    ERROR "Column not found"
    GOTO ent_col
END IF
...
```

This source code sample makes the program ignore errors, if any, but is purposed for correcting one of the possible exceptions. If the user inputs a value which does not match any of the names of the columns that comprise the current table, an error will occur during the *take_values* function execution, and the user will be asked to enter the column name once more.



Fatal Error Handling

Some errors which occur during the program execution are fatal and prevent the program from execution. If the error is fatal and the application cannot continue, the program will be terminated, even if you specify CONTINUE. However, 4GL offers the tools (i.e. CALL and GOTO actions) which allow you to handle the error and make sure that no data is lost or/and to produce the message for the user before the program terminates.

The CALL and GOTO actions are executed, even if 4GL encounters a fatal error. However, the program is terminated afterwards. The STOP action lets the program terminate without handling the error.

The Error Scope

The term error scope is used to indicate whether the program will handle the errors that occur during the 4GL expressions processing. 4GL supports the normal error scope and the AnyError Scope.

Normal error scope means that the program will handle SQL errors, screen interaction statement errors, validation errors that occur during the execution of the VALIDATE statement, but will ignore the errors that occur during the processing of other 4GL expressions. The normal error scope is the default one and does not need to be set specially. If you want the ERROR keyword to be of a normal scope, you can also use the SQLERROR keyword instead of ERROR; they are synonymous within the WHENEVER statement.

The NOT FOUND Event

The NOTFOUND keyword indicate an event which checks for the end of the data which typically occurs when the FETCH statement goes beyond the last line of the currently active set, or when the SELECT statement returns no rows. If the NOT FOUND condition is specified in the WHENEVER statement, the program treats the SELECT and FETCH statements differently from other SQL statements:

The program checks whether the FETCH statement has retrieved a row which is above the first or below the last row of the active set;

The program checks whether the SELECT statement returns no rows.

If both conditions are satisfied, the sqlca.sqlcode variable gets the value 100, which means not found. The following statement calls the function check_querry if the NOT FOUND condition is satisfied:

```
WHENEVER NOT FOUND CALL check_querry
```



Note: Do not confuse the NOT FOUND condition of the WHENEVER statement with the NOTFOUND keywords in status test (e.g., IF status = NOTFOUND, THEN...). Although these keywords are synonymous and they have the same meaning, their spelling is not interchangeable.

The WARNING Event

You can use the WARNING (or SQLWARNING) event to make the program perform an action when an SQL statement generates a warning. For example, the following statement passes the program control to a specified place of the program whenever a warning occurs:

```
WHENEVER WARNING GOTO checkpoint
```



Warnings do not affect the execution of the program but are recorded in the corresponding built-in variables, unless you handle the warning explicitly. If a warning occurs, the values of the sqlca.sqlwarn built-in array are changed. This array is described in detail later in this chapter together with the reasons for the warning occurrence.

The ARGUMENT ERROR Event

If the WHENEVER ARGUMENT ERROR statement is in effect, 4GL takes the specified action when it encounters that the incorrect number or type of variables are passed to a function call.

```
WHENEVER ARGUMENT ERROR CONTINUE
```

The RETURN ERROR Event

If the WHENEVER RETURN ERROR statement is in effect, 4GL takes the specified action when incorrect parameters are returned from a function to the calling routine.

The UNDEFINED FUNCTION Event

If the WHENEVER UNDEFINED FUNCTION statement is in effect, 4GL takes the specified action when it encounters a function call which function has not been defined.

Typically the compiler verifies the functions called within an application to make sure that they have been defined, thus an undefined function will produce a compile-time error in the majority of cases. However, there are two situations in which an undefined function can pass through compilation process and remain undefined at runtime:

- When an application is linked in Lycia Command Line environment and the qlink command has been used without the -verify flag
- When an application is linked against a dynamic library. The function must be defined at the time of linking, however, if the dynamic library is changed later and the function is changed or removed, it will not be reflected in the application, unless you recompile it again

The QUIT Event

The WHENEVER QUIT command is an alternative to the DEFER QUIT statement which provides more flexibility than the DEFER statement. The DEFER QUIT statement effectively translates to:

```
WHENEVER QUIT CALL EventQuitFlag
```

The WHENEVER statement allows you to modify the 4GL reaction to the Quit key being pressed.

The INTERRUPT Event

The WHENEVER INTERRUPT command is an alternative to the DEFER INTERRUPT statement which provides note flexibility than the DEFER statement. The DEFER INTERRUPT statement effectively translates to:

```
WHENEVER INTERRUPT CALL EventIntFlag
```

The WHENEVER statement allows you to modify the 4GL reaction to the Interrupt key being pressed



The TERMINATE Event

If the WHENEVER TERMINATE statement is in effect, 4GL takes the specified action when the process is being terminated. This option takes effect only if the process is running on an application server.

The HANGUP Event

If the WHENEVER HANGUP statement is in effect, 4GL takes the specified action when the connection to the client or terminal has closed. This option takes effect only if the process is running on an application server, i.e. only in GUI mode.

The FATAL ERROR Event

If the WHENEVER FATAL ERROR statement is in effect, 4GL takes the specified action when a fatal application error has occurred (i.e. segmentation fault).



Note: When the FATAL ERROR event is returned, the program will always exit. This does, however, provide some simple capability to report/clean up for such conditions.

The Actions Applied to the Trapped Events

As has already been said, the action in the WHENEVER statement stands for an action to be performed if the specified event occurs. The action can be represented by such statements as CONTINUE, GOTO, STOP, and CALL.

The GOTO Statement

The GOTO statement, used together with the label-name, passes the program control to the labelled place of the source code, if the specified error or warning takes place.

The CONTINUE Statement

The CONTINUE statement makes the program perform no action, if the specified condition is met and prevents the program from termination. The CONTINUE action is the default action for all the conditions and is applied when any of the conditions (except ERROR) is satisfied and no other actions are specified for them in the WHENEVER statements.

The STOP Statement

The STOP statement forces the program termination, if the specified condition is satisfied.

The event and the action clauses are obligatory and each of them can include only one item. The item that comprises a clause must be one of those listed above.

The RAISE Statement

The RAISE action is typically specified in function blocks. After the RAISE statement is encountered by the program, any error occurring in the function will be passed to the calling function. If this function does not contain an error handler, the error will be passed to the parent of this function, etc (see TRY...CATCH statement described below).

However, the RAISE statement is not executed when used within a REPORT program block.



The CALL Statement

The CALL statement is used to transfer program control to a specified function. It is impossible to pass values to the function called within the WHENEVER statement, so you must not use the parentheses after the function name (e.g.: WHENEVER ERROR CALL my_function).

Here we should give the definition of an event handler, because it is an important notion for Querix 4GL. An event handler is a programmer-defined or built-in 4GL function with no parentheses and no argument list, which, as its name suggests, is used to handle certain events. EventInterruptFlag in the following code line is one of such handlers.

```
WHENEVER ERROR CALL EventInterruptFlag
```

Querix supports some special event handlers used with the CALL option. They are listed in the table below.

Function	Effect
EventAbort	The process is aborted with an error dialog.
EventTerminate	The program exits immediately with no error.
EventIgnore	The event is ignored and the process is continued. This option will retry all input statements in process rather than executing them like the WHENEVER... CONTINUE statement.
EventInterruptFlag	Sets the int_flag to a non-zero value and continues.
EventQuitFlag	Sets the quit_flag to a non-zero value and continues.
EventTerminateDialog	Displays a dialog box requesting permission to terminate. If the user selects yes, the program terminates.

Default Event Handlers

The system installs default global-level event handlers at start-up which take effect when the corresponding event handlers are not specified explicitly. These are as follows:

Event	Default handler
ARGUMENT ERROR	EventAbort
RETURN ERROR	EventAbort
UNDEFINED FUNCTION	EventAbort
INTERRUPT	EventTerminate
QUIT	EventTerminate
TERMINATE	EventTerminate
HANGUP	EventTerminate
FATAL ERROR	EventTerminate
ERROR	EventAbort
WARNING	EventAbort

User Event Handlers

In addition to the standard processing functions the user may write event processing routines. Such routines always return a value. If they return nonzero, then the routine is considered not to have processed the event and it is passed to the next level handler. Therefore you can write:



```
FUNCTION work()
WHENEVER TERMINATE CALL mycleanup
INSERT INTO mywork VALUES (my_session_id,'an entry')
END FUNCTION

FUNCTION mycleanup()
DELETE FROM mywork WHERE id = my_session_id
RETURN 1
END FUNCTION
```

This would cause the mycleanup function to be called if the program receives the terminate signal.

The Scope of the WHENEVER Statement

The WHENEVER statement can be used only within a program block, it cannot be used outside a program block, and it cannot be used in the GLOBALS block either. So, it has the module scope of reference. The WHENEVER statement is in effect from the place where it is activated in the program till the next WHENEVER statement which includes the same condition. If there is no other WHENEVER statement with the same condition, the previously specified action is in effect until the end of the module. The following snippet of source code demonstrates how the WHENEVER statement can be used for several times in one program:

```
MAIN
...
WHENEVER NOT FOUND GOTO inp_user -- makes the program respond
                                -- to the NOT
                                -- FOUND condition
...
LABEL inp_user:
PROMPT "Enter the user name " FOR us_name
CALL check_data(us_name) -- if the us_name has no value,
                        -- the program will
                        -- return the user to the PROMPT
                        -- statement due
                        -- to the WHENEVER statement above

...
WHENEVER NOT FOUND CONTINUE -- the error handling principle changes here
END MAIN
```

The TRY... CATCH statement

The TRY...CATCH statement is another statement useful in error handling. The general syntax of the statement is as follows:

```
TRY try_block [CATCH catch_block] END TRY
```

When the program encounters the TRY keyword, it begins to execute the statements specified in the *TRY block*. These can be any 4GL or SQL statements. If any exception occurs during the statements execution, the program control passes to the *CATCH block* which contains the statements that are to be executed in case of the error occurrence.



The execution of the source code below must result in a table creation. If the table already exists, an error will occur. In this case, the CATCH block opens a dialog window which allows user to choose whether to use the existing table or delete it.

Note, that the TRY statement is not a loop, so if you want the program to execute the statements from the TRY block after the error is handled, you have to add corresponding commands to the CATCH block.

```
TRY
    CREATE TABLE works_list
    (
        obj_title      VARCHAR(100),
        object         BYTE,
        obj_descr      VARCHAR(200),
        obj_create_date DATE,
        obj_info       VARCHAR(200)
    )
    DISPLAY "Table created" AT 2,2

CATCH
    LET answer = fgl_winquestion("Table already exists",
                                  "Do you want to delete the existing one?",
                                  "No", "Yes|No", "Question", 1)
    IF answer = "Yes"
        THEN DROP TABLE works_list
    # After the table is dropped, we can create a new one
        CREATE TABLE works_list
        (
            obj_title      VARCHAR(100),
            object         BYTE,
            obj_descr      VARCHAR(200),
            obj_create_date DATE,
            obj_info       VARCHAR(200)
        )
        DISPLAY "Table created" AT 2,2

    END IF
```



```
END TRY
```

If no CATCH block is specified in the statement, the program control goes to the statement following the END TRY keywords in case of an exception.

If all the statements in the TRY block are executed without exceptions, the program execution continues from the statement following the END TRY keywords.

TRY...CATCH statement can be effectively used together with the RAISE action of the WHENEVER statement. You can place the CALL statement to the TRY block and the RAISE option to the invoked function. If any error occurs during the function execution, the program control will pass to the CATCH block:

```
MAIN
TRY
    CALL do_exception(100, 0)
CATCH
    CALL foo()
END TRY
END MAIN

FUNCTION do_exception(a, b)
    DEFINE a, b, c INTEGER
    WHENEVER ANY ERROR RAISE
    RETURN a / b --
END FUNCTION

FUNCTION foo()
    DISPLAY "Exception caught, status: ", STATUS
END FUNCTION

It is also possible to nest a TRY statement into another TRY statement, if needed:
TRY
    SELECT * INTO myrec.* FROM tab_1 WHERE col1 = 1
CATCH
    TRY
        DATABASE default_db
        SELECT * INTO myrec.* FROM tab_1 WHERE col1 = 1
    CATCH
        ERROR "An error occurred during the SELECT statement execution"
    END TRY
END TRY
```



The TRY block can be compared with the WHENEVER ANY ERROR GOTO *label*/construction. The execution of the extract given below will have the same results as that of the first example given in this section:

```
WHENEVER ANY ERROR GOTO catch_error

CREATE TABLE works_list
(
    obj_title      VARCHAR(100),
    object         BYTE,
    obj_descr      VARCHAR(200),
    obj_create_date DATE,
    obj_info       VARCHAR(200)
)
DISPLAY "Table created" AT 2,2

GOTO no_error
LABEL catch_error:

LET answer = fgl_winquestion("Table already exists",
    "Do you want to delete the existing one?",
    "No", "Yes|No", "Question", 1)
IF answer = "Yes"
    THEN DROP TABLE works_list
END IF

LABEL no_error:
...
```

Error Handling Using the SQLCA Record

Above, we have mentioned the built-in variable sqlca which is used to store information about the success of execution of the most recent SQL statement.

When 4GL executes an SQL statement, it returns a set of values to the sqlca record members. The values are changed after each SQL statement execution and describe different aspects of the operation performance.

The sqlca record is built-in and mustn't be declared by the programmer. If you declare a variable with the same name you will get a compile-time error. The structure of this built-in record is as follows (you do not need to include this code into your application to be able to use this variable):



```

sqlca RECORD
    sqlcode INTEGER,
    sqlerrm CHAR(71),
    sqlerrp CHAR(8),
    sqlerrd ARRAY [6] OF INTEGER,
    sqlwarn CHAR(8)
END RECORD

```

The meanings of the sqlca record members are described in the table below:

MEMBER NAME	DESCRIPTION
sqlcode	Indicates the success of any SQL statement execution. Zero value means that the statement has been executed successfully. Value 100 (NOT FOUND) means that the statement has been executed successfully, but has returned null. The value is negative, if the statement execution fails.
sqlerrm	Is not used currently
sqlerrp	Is not used currently
sqlerrd	An array of the INTEGER data type which includes 6 members
sqlerrd[1]	Is not used currently
sqlerrd[2]	Represents the SERIAL value or an ISAM error code
sqlerrd[3]	Represents the number of rows that have been inserted or changed. If a multi-row UPDATE or INSERT statement fails, the value represents the number of rows that have been processed successfully. The value is set to 1 if a LOAD statement fails and the error ID is -846 (number of values in the load file is not equal to the number of columns).
sqlerrd[4]	Describes the estimated CPU resources that will be necessary for the query performance
sqlerrd[5]	Represents the offset of the error that occurred during the SQL statement execution
sqlerrd[6]	Indicates the row number of the last row that was updated by the most recent statement. This value may be returned or not; it depends on the database server.
sqlwarn	Is represented by an 8-character string whose characters indicate the warnings that result from the SQL statement execution. If no problems occur, the character string remains blank.
The rest of the table explains each character of this character string. Each character may be referenced as character substrings containing exactly one character.	
sqlwarn[1,1]	The first character gets value w, if one or several of the other characters are set to w. This character is blank only if all the other characters are blank too.
sqlwarn[2,2]	The second character gets value w, if the program has truncated one or more values in order to make them fit into a CHAR variable. It also gets w value, if a program has executed the DATABASE statement to specify a database which has transactions.
sqlwarn[3,3]	The third character gets value w, if the program uses the null value when evaluating the SUM(), AVG(), MAX, or MIN() functions, or if the DATABASE statement selects a database which is ANSI-compliant.
sqlwarn[4,4]	The fourth character gets value w, if the number of program variables in the INTO clause of the SELECT statement doesn't match the number of items in the select-list of the SELECT clause. It is also set to w, if the DATABASE statement indicates an Informix Dynamic Server database.
sqlwarn[5,5]	The fifth character gets the value w, if a float number is converted into a decimal number.
sqlwarn[6,6]	The sixth character is evaluated to w, if the SQL syntax is extended to the ANSI/ISO standard.
sqlwarn[7,7]	The seventh character is evaluated to w, if a query skips a fragment of the table or if the



database locale differs from the client's one.
sqlawarn[8,8] Is not used currently

You can use the WHENEVER statement and refer to these variables in order to specify different actions for different kinds of errors and warnings that can occur during the program execution:

```
...
WHENEVER ANY ERROR CONTINUE
CALL func_1() RETURNING my_arr
IF sqlca.sqlerrd[3] < 3 THEN -- checks the number of rows updated by
-- the function func_1()
MESSAGE "The information is not enough"
END IF

IF sqlca.sqlcode < 0 THEN -- checks if there has occurred any error
GOTO retry
END IF
```

Remember, that the WHENEVER ERROR CONTINUE statement is necessary if you plan to refer to status, sqlca.sqlcode and sqlca.sqlerrd variables which may have values representing error codes. Otherwise, the program will be interrupted as soon as an error is detected.

Retrieving the Error Message

Q4GL supports a number of built-in functions for error handling. With their help you can retrieve a particular error text message or an code number, create, open and fill a special error log file with information, and so on. Some of these functions are described in the following sections.

To get the text of an error message with the error code specified three functions can be used in Q4GL: *err_get()*, *err_print()* and *err_quit()*.

They all take a single argument – an integer that represents a negative value of the *status* built-in variable for both 4GL and SQL errors or the member of the sqlca global record – *sqlcode* for SQL errors only, and return a character string - the text message corresponding to that value.

```
err_get (status / sqlca.sqlcode)
err_print (status / sqlca.sqlcode)
err_quit (status / sqlca.sqlcode)
```

The code returned by the *status* variable or the *sqlcode* record member can be used literally as the functions argument:

```
CALL err_get(-1110)
```

It can also be assigned to a variable of the INTEGER data type:

```
...
DEFINE err_code INT
LET err_code = status
CALL err_print(err_code)
...
```

Each of these functions, however, has its own peculiarities.

So, the *err_get()* function requires a special CHAR variable to store the text string returned by the function, if it needs to be displayed to the screen. In the following code snippet the value that the function returns is



first assigned to the *error_text* variable, and then displayed at the last line of the screen by means of the ERROR statement.

```
...
DEFINE error_text CHAR (100), error_code INT
...
LET error_code = -1110
LET error_text = err_get(error_code)
# Below is the alternative way to invoke the function
# CALL err_get(error_code) RETURNING error_text
ERROR error_text
...
```

Function *err_print()* , in its turn, automatically displays the error message, and you don't have to include the ERROR statement into your code. But if you do, the message will not be visible.

Function *err_quit()*, unlike the other two, returns the error message and terminates the program right after the error message has been displayed.

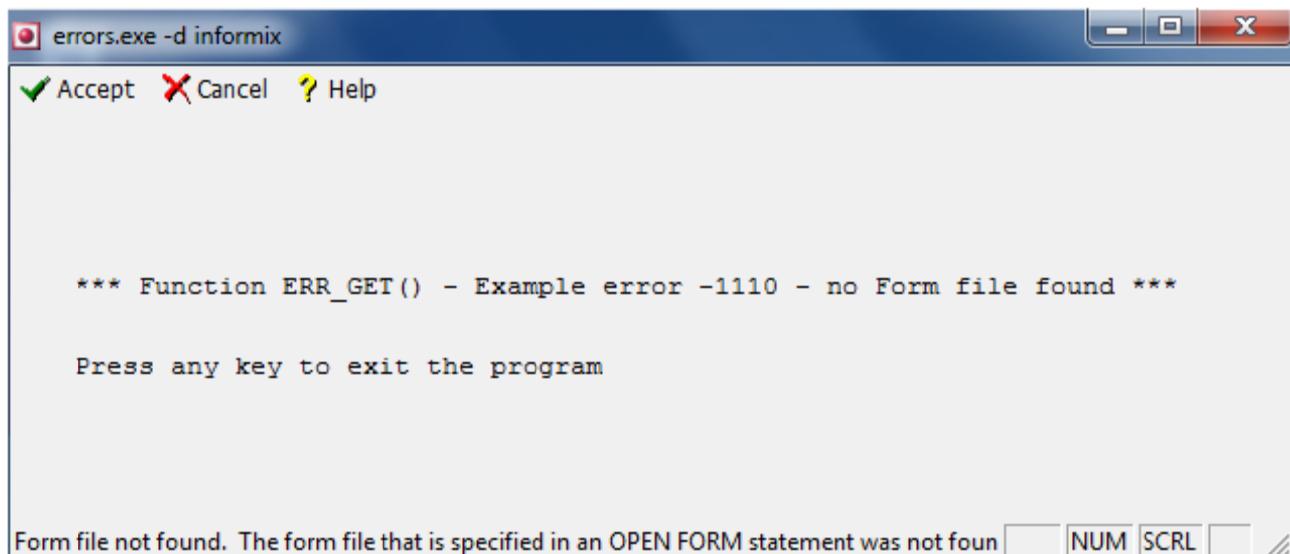


Note: The *err_quit()* function currently works only in the character mode.

The example below illustrates the *err_get()* function application, although the same code with certain modifications can be used to demonstrate the work of the other two functions.

- 1110 - the value of the global *status* variable, is assigned to the variable *current_error_status*. After the program is launched, the message corresponding to the error number will be displayed at the bottom of the screen, as can be seen in the following screenshot.

```
MAIN
DEFINE
current_error_status, num INTEGER,
my_error_text CHAR(100)
WHENEVER ERROR CONTINUE
OPEN WINDOW win1 WITH FORM "my_form" -- we try to open a non-existing form
-- which will set status to -1110
LET current_error_status = status
IF current_error_status < 0 THEN
LET my_error_text = ERR_GET(current_error_status)
ERROR my_error_text
END IF
DISPLAY "Press any key to continue" AT 2,2
LET num = fgl_getkey()
END MAIN
```



Working with an Error Log File

The information about errors the program encounters during its execution can be written into a special error log file. This way you can record the exact date and time when a particular error was produced, its number, the code line where it was discovered, the error message text. This data may prove to be very useful for enhancing the work of your application as well as for tracking down any attempts at the breach of security.

Opening or Creating an Error Log File

To create a new error log file or to open an existing one, the built-in function *startlog()* is used.

```
startlog ("[path]filename")
```

The function takes only one argument – *filename*, which is a character string representing the name of the error log file. If this file doesn't exist, it will be automatically created in the program folder.

The filename must be quoted and have an extension. If the error log file is not located in the program folder, you must include the path to it. This line of code specifies the location of the "*errors.txt*" log file on drive C in the folder called "err":

```
startlog("C:/err/errors.txt")
```

Alternatively, the file name can be assigned to a variable of the CHAR or VARCHAR data types and used as the function argument instead of a character string.

```
LET file_name = "errorfile.txt"  
startlog (file_name)
```

By default the information to be written into the newly created or already existing error log file will have the following format:

```
Time: yyyy-mm-dd hh:mm:ss(3), User: user_name  
An exception (error) is being handled.  
Module: module_name  
File: file_name, line line_number
```



```
Function: function_name
The error code (error_code) was received.
Text: error_text_message.
```

Here *yyyy-mm-dd hh:mm:ss(3)* stands for the date and time when the particular error was produced. *user_name* is the name of the current database user. *module_name* is the name of the program, *file_name* is the name of the source file with an extension, *line_number* is the line in the program code that contains the error, *function_name* represents the name of the function with the error, *error_code* is a negative value returned by the global status variable or the sqlca record member – sqlcode, and finally, *error_text_message* is the actual text of the error message.

Below is a typical entry in an error log file which follows this format.

```
Time: 2010-10-29 13:49:09.316, User: Unknown
An exception (error) is being handled.
Module: contracts.exe [000000b1]
File: contr.4gl, line 10 (1-9)
Function: main
The error code (-206) was received.
The specified table <table-name> is not in the database...
```

Every time the program encounters an error, it automatically adds a record about it to the error log file after the last entry.

The code below illustrates how the *startlog()* function is applied in practice. First, we call the function and specify the new log file name as its argument. We follow it by a SELECT statement to retrieve data from a non-existent table. After the program is run, the corresponding error is produced and the application is terminated immediately. Then, if you open the "errors.txt" file, it will contain one record in the format described above. This can be seen in the screenshot after the example.

```
DATABASE db_examples

MAIN

CALL startlog("errors.txt")

SELECT * FROM tabb1

END MAIN
```



The screenshot shows a Windows Notepad window titled "errors.txt - Notepad". The content of the window is a text message about an exception being handled. The text reads:

Time: 2010-10-29 15:23:48.189, User: UnknownAn exception (error) is being handled.Module: errors.exe [000000ac]File: errorsfile.4gl, line 8 (1-20)Function: mainThe error code (-206) was received.The specified table <table-name> is not in the database.The database server cannot find a table or view specified in the statement.The table or view might have been renamed or dropped from the database.You might also get this message if you omit the keyword "TYPE" when you are trying to grant USAGE privileges on a user-defined type. For example, the following GRANT statement is correct: GRANT USAGE ON TYPE person_row_type TO usr2;The following GRANT statement, however, generates error -206: GRANT USAGE ON person_row_type TO usr2;Check the names of tables and views in the statement or check for omission of the keyword "TYPE" in a GRANT statement. If the names are spelled as you intended and "TYPE" is not missing, check that you are using the database you want. To find the names of all tables in the database, query the systables table. To find the names of all views, query the sysviews table.

Adding Comments to the Error Log File

As we have already explained, when the program finds an error and the *startlog()* function is invoked, a record related to this error is created in a strictly specified format. In many cases, however, you may want to add your own comments about the exceptions the program generates. For this purpose a special built-in function has been implemented in Q4GL – *errorlog()*.

It takes one argument – a quoted character string which contains the text you want to be inserted into the error log file.

```
errorlog ("text_message")
```

The text string can be assigned to a variable of the CHAR data type:

```
...
DEFINE text_mes CHAR (100)
...
LET text_mes = "The first error"
CALL errorlog (text_mes)
...
```

The *errorlog()* function is normally used in conjunction with the *startlog()* and *err_get()* functions, which is demonstrated by the following example.

```
DATABASE db_examples

MAIN

DEFINE err_mes CHAR (255)

CALL startlog ("errors.txt") -- we open or create the log file

WHENEVER ERROR CONTINUE      -- we prevent the termination

SELECT * FROM tabbb1          -- this statement selects from a non-
                                -- existing table

IF status < 0 THEN -- if the error took place

LET err_mes = err_get (status)-- the value returned by the err_get
                            -- function is assigned to the
```



```
-- variable err_mes

CALL errorlog ("First error")-- the errorlog() function is called
-- to add a comment to the message

CALL errorlog (err_mes)      -- the standard error description is
-- added to the error log file
END IF

END MAIN
```

Now if you open "*errors.txt*" you will see the comment "First error" added to the record about the error.

Accompanying an Event with a Sound Signal

To accompany an event with a sound, for example, to warn the user of some errors, or to prompt them for some actions, you can use the *fgl_bell()* function. It takes no arguments and simply invokes the system bell, which is a short bell sound, to the client.

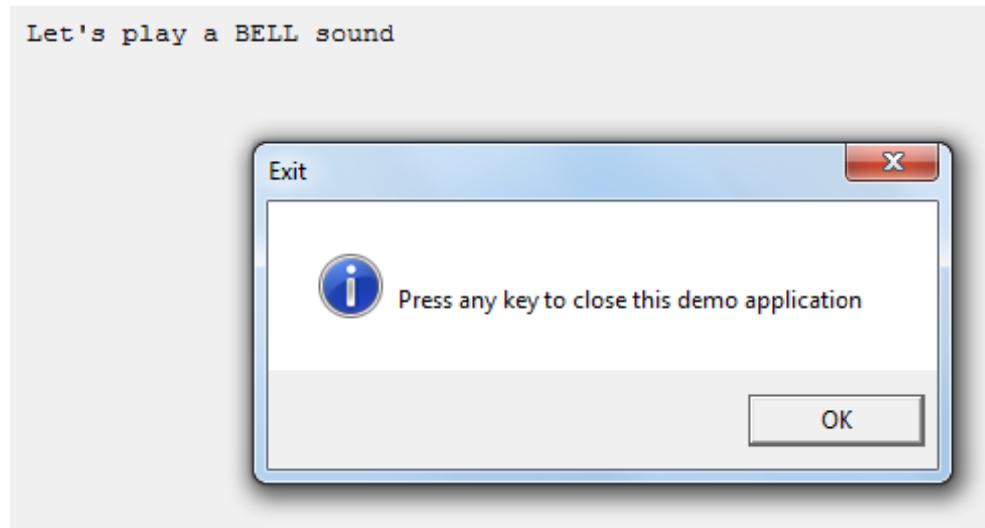
```
fgl_bell()
```

The sound is played only once upon each event occurrence. If you want it to be played a number of times, for example upon several function calls, you have to invoke the *fgl_bell()* function for each subsequent event.

When the following code is executed, the user will hear a short bell sound and see a dialog window prompting them to close the application.

```
MAIN
DISPLAY "Let's play a BELL sound" AT 5,5
CALL fgl_bell()
CALL fgl_winmessage("Exit","Press any key to close this demo
application","info")
END MAIN
```

You can change the order in which the functions in this example are invoked. Then the sound will be played once you have pressed the OK button.



Retrieving Errors from a Non-Informix RDBMS .

When you are using a RDBMS other than Informix and the system encounters an error, Q4GL tries to provide you with the closest Informix error code and a corresponding text message on a particular error condition. However, you might be interested in the native RDBMS error code and message rather than in their Informix equivalent. In this case, you can make use of two special functions implemented in Q4GL for this purpose: *fgl_native_code()* and *fgl_native_error()*

Getting an Error Code from a Non-Informix RDBMS

The *fgl_native_code()* function requires no arguments, and returns a value of the INTEGER data type which represents the actual error code from a native RDBMS for the last SQL statement executed as opposed to the Informix error code.

The example below illustrates the function application. The database management system employed by the application has been changed from the INFORMIX RDBMS to a different one. The native error code retrieved by *fgl_native_code()* is assigned to the variable "*native_code*" and then displayed to the screen. The output can be seen in the figure following the code.

```
DATABASE test2
MAIN
DEFINE sql_stmt VARCHAR(255)
DEFINE native_code INTEGER
LET sql_stmt = "Invalid SQL statement"
WHENEVER ERROR CONTINUE
PREPARE p1 FROM sql_stmt
DECLARE c1 CURSOR FOR p1
OPEN c1
WHENEVER ERROR STOP
LET native_code = fgl_native_code()
DISPLAY "" AT 7, 5
DISPLAY "Native error code: ", native_code AT 16,5
CALL fgl_winmessage("Exit",
```



```
"Press any key to close this demo application", "info" )  
END MAIN
```

Retrieving the Text of a Native RDBMS Error Message

The purpose of the *fgl_native_error()* function is to retrieve the text of an error message produced by a native database management system for the last SQL statement executed.

```
fgl_native_error( )
```

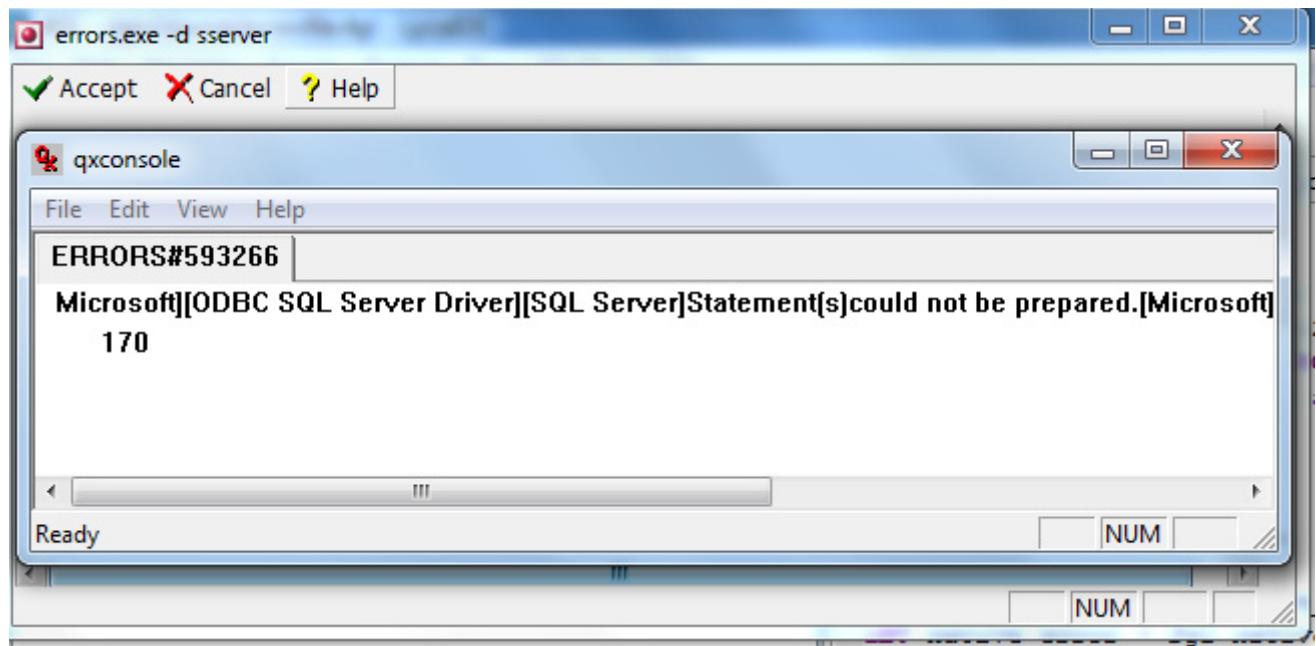
The function requires no arguments and returns a CHAR value which represents the text of an error message generated by a native RDBMS.

The returned value can be assigned to a variable of the CHAR data type:

```
LET error_mes = fgl_native_error( )
```

The following example includes both the *fgl_native_code()* function and the *fgl_native_error()* function. Now the user will see both the code and the text of an error message produced by the native RDBMS.

```
DATABASE test2  
MAIN  
DEFINE sql_stmt VARCHAR(255), native_error VARCHAR(255),  
native_code, num INTEGER  
LET sql_stmt = "Invalid SQL statement"  
  
WHENEVER ERROR CONTINUE  
PREPARE p1 FROM sql_stmt  
DECLARE c1 CURSOR FOR p1  
OPEN c1  
WHENEVER ERROR STOP  
  
LET native_code = fgl_native_code()  
LET native_error = fgl_native_error()  
  
DISPLAY native_error  
DISPLAY native_code  
  
LET num = fgl_getkey()  
END MAIN
```



Finding out the Last Error Returned by the Database Interface

The `fgl_driver_error()`. function is called when you need to know the last error produced by the Lycia database interface. It doesn't need any arguments and returns a character string, which is the text message of a particular Lycia database interface error.

In the following example a window containing two error messages is invoked when the program is launched. The first error is generated by the Lycia database interface, while the other - by the native RDBMS. The result of the code execution can be seen in the screenshot following the code.

```
DATABASE test2

MAIN

DEFINE sql_stmt VARCHAR(255), driver_error, native_error VARCHAR(255),
num INT

LET sql_stmt = "Invalid SQL statement"

WHENEVER ERROR CONTINUE

PREPARE p1 FROM sql_stmt
DECLARE c1 CURSOR FOR p1
OPEN c1
```

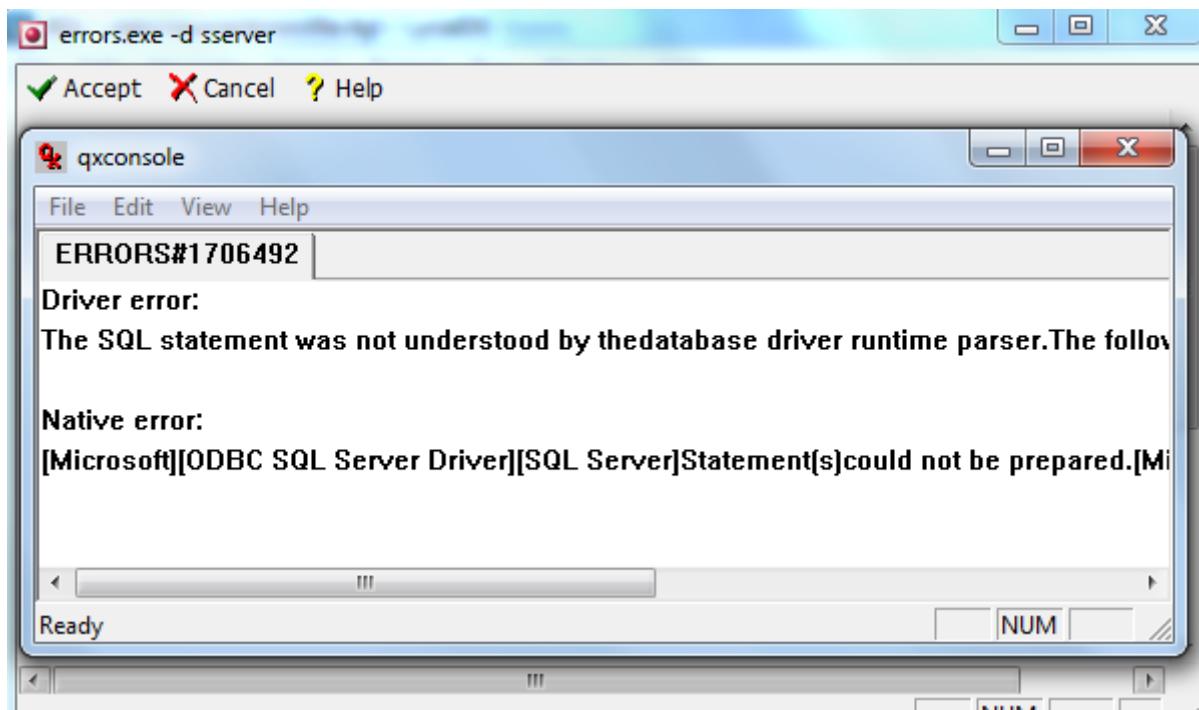


```
WHENEVER ERROR STOP

LET driver_error = fgl_driver_error()
LET native_error = fgl_native_error()

DISPLAY "Driver error: "
DISPLAY driver_error
DISPLAY ""

DISPLAY "Native error: "
DISPLAY native_error
LET num = fgl_getkey()
END MAIN
```



Example

This example illustrates how you can handle different types of errors and warnings using the 4GL syntax. It intentionally contains some events which will generate errors or warnings which are then handled.

For this example to function you need to have the connection to the database specified in the DATABASE statement at the beginning of the module. You should also run the example from [Databases in 4GL](#) chapter



and from [Populating Database Tables](#) chapter to create the necessary tables and to populate them with values, if they do not yet exist.

```
#####
# This applications handles errors using the WHENEVER statement,
# as well as the built-in variables status and SQLCA.
#####
DATABASE db_examples
DEFINE
    var_date DATE,
    r_clients     RECORD LIKE clients.*,
    r_contracts   RECORD LIKE contracts.*

MAIN
DEFINE

    var_smallint    SMALLINT,
    var_char,        CHAR(80),
    error_msg,      CHAR(80),
    v_status,       INTEGER,
    i_row, i_col,   INTEGER,
    v_sqlawarn,    CHAR(1),
    error_code,    INTEGER,
    w_name,         VARCHAR(15),
    at_line,
    at_column,
    with_rows,
    with_columns    SMALLINT

OPTIONS
    INPUT WRAP

    # After we call STARTLOG function the message about any error that might
    # occur will be passed to the log file called "error.log".
    # If there is no such file already existing in your system, it will be
    # created.
    # After the program is terminated, you can view this file,
    # To see what kinds of errors occurred during the program execution.
    CALL STARTLOG("error.log")

DECLARE c_sel_clients CURSOR FOR -- we declare the cursor we will use later
                                -- in the program
    SELECT c.* FROM clients c

MENU "Errors"

COMMAND "Non-Handled Conversions"
        "Not handled errors in calculations and data conversion"

CLEAR SCREEN
WHENEVER ERROR CONTINUE
    # We will use several LET statements which will assign invalid values
    # or cause impossible data conversion. These errors will not be handled
```



```
# and the results will be displayed to the screen

# We assign a value which is greater than the maximum value for the
# SMALLINT
LET var_smallint = 32767 + 3
DISPLAY " LET var_smallint = 32767 + 3:      var_smallint = ",
var_smallint    AT 4,1

# We assign an invalid date to var_date variable of the DATE data type
LET var_date = "13/32/2011"
DISPLAY " LET var_date = ""13/32/2011"":           var_date = ",
var_date        AT 6,1

# We assign a number with a character to a numeric variable
LET var_char = "12345a"
LET var_smallint = var_char
DISPLAY " LET var_smallint = ""12345a"":       var_smallint = ",
var_smallint AT 8,1

WHENEVER ERROR STOP

COMMAND "Handled Conversations"
"Handling errors in calculations and conversion with WHENEVER ANY ERROR"

CLEAR SCREEN

# The statement WHENEVER ANY ERROR traps SQL errors, screen interaction
# errors, errors of initialization and validation, as well as the
# errors of any calculations and conversions in 4GL.

# This statement causes the program to terminate,
# if either 4GL or SQL error occurs.
WHENEVER ANY ERROR STOP

DISPLAY "WHENEVER ANY ERROR CONTINUE: var_smallint = 32767 + 3"
      AT 4,2 ATTRIBUTE(GREEN, BOLD)

# This statement forces the program to continue regardless of the error
WHENEVER ANY ERROR CONTINUE -- if you comment this statement, the
                                -- program will produce an error

LET var_smallint = 32767 + 3

LET v_status = status -- we assign the value of the built-in variable
                      -- to v_status to track error numbers

# We compose the error message using the err_get() function to retrieve
# the error message corresponding to the status value
LET error_msg = " status = ",v_status USING "-<<&",
" . ",ERR_GET(v_status)

# We truncate the error message, to be sure it is not wider than the
# screen
LET error_msg = error_msg[1,53]
# Then we display this message to the screen
```



```
DISPLAY error_msg AT 5,1 ATTRIBUTE (RED,BOLD)

LET var_smallint = 0 -- we assign a valid value to the variable
DISPLAY " We assign a valid value after an error occurred: ",
        var_smallint USING "<<<&","." AT 6,1
DISPLAY " var_smallint = ",var_smallint AT 7,1

WHENEVER ANY ERROR STOP -- we set the error handling behaviour to the
                        -- default state

DISPLAY "WHENEVER ANY ERROR CALL: var_date = ""13/32/2011"""
        AT 9,2 ATTRIBUTE(GREEN, BOLD)

# Using this statement we handle the error using a programmer defined
# handler function which is declared at the end of this module
WHENEVER ANY ERROR CALL my_error_handler

# Here we assign an invalid date which triggers the function
# specified in the WHENEVER statement
LET var_date = "13/32/2011"

DISPLAY " var_date = ",var_date AT 12,1

WHENEVER ANY ERROR STOP

DISPLAY "WHENEVER ANY ERROR GOTO: var_smallint= '12345a' "
        AT 14,2 ATTRIBUTE(GREEN, BOLD)

# This statement forces the program to continue after an error
# and passes the program control to the specified label
# if an error occurs
WHENEVER ANY ERROR GOTO lab_1 -- Comment this statement, recompile and
                            -- run the program to see how it will
                            -- react.

LET var_char = "12345a"      -- we assign the valid value to a character
                             -- variable
LET var_smallint = var_char -- and then reassign it to the numeric
                           -- variable where it is invalid

WHENEVER ANY ERROR STOP

LABEL lab_1:                  -- here is where the program control is
                            -- passed
LET v_status = status         -- then we find out the error code and
                            -- error message
LET error_msg = " status = ", v_status USING "<<<&",
".",ERR_GET(v_status)
LET error_msg = error_msg[1,65]
DISPLAY error_msg AT 15,1 ATTRIBUTE (RED,BOLD)
LET var_smallint = 12345      -- we reassign the valid value to continue
                            -- the program execution
DISPLAY " We assign a valid value after an error occurred: ",
        var_smallint USING "<<<<&","." AT 16,1
DISPLAY " var_smallint = ",var_smallint AT 17,1
```



```
WHENEVER ERROR STOP

COMMAND "Coordinates Errors"
    "Handling the errors caused by wrong coordinates for objects"

CLEAR SCREEN

# This statement continues the execution of the program if an SQL error,
# screen interaction error, validation or initialization error occurs.
WHENEVER ERROR CONTINUE -- Comment this statement, recompile and run
                           -- the program to see how it will react.

CALL fgl_window_open("w_qry_clients", 4,4, "err_window", true)

DISPLAY !" TO quit_but
DISPLAY !" TO open_w

INPUT BY NAME w_name, at_line, at_column, with_rows, with_columns
BEFORE INPUT
    LET var_char = " This text is displayed outside the window border"
    LET i_row = 25
    LET i_col = 81

# The DISPLAY statement below causes an input/output error, because
# it displays information to a location outside the screen (25, 81).
# As the result status variable acquires value -1135 which is the error
# code. The execution of the program will continue due to the WHENEVER
# statement above.
    DISPLAY var_char CLIPPED, " " AT i_row,i_col ATTRIBUTE (YELLOW,BOLD)
    LET v_status = status
    LET error_msg = "status = ",v_status USING "-<<&",". ",
                  ERR_GET(v_status)

    ERROR error_msg ATTRIBUTE (RED,BOLD)
    CASE
        WHEN v_status < 0 -- if an error occurred
            -- we handle this error, specifying the correct coordinates
            -- for the DISPLAY statement
            -- And display the information once more
            LET i_row = 10
            LET i_col = 40-LENGTH(var_char)/2
            DISPLAY var_char CLIPPED, " " AT i_row,i_col
                  ATTRIBUTE (YELLOW,BOLD)

        WHEN v_status = 0 -- this will happen if the program is run in GUI
                          -- because the GUI clients can accept coordinates
                          -- outside the current window
            CALL fgl_winmessage("Coordinates",
                               "You can drag the window borders to see the string" ||
                               "\ndisplayed outside the window borders.", "info")
    END CASE
```



```
ON KEY (F10)
    EXIT INPUT
AFTER INPUT
    # We open the window with the inputted parameters
    # If youi remove the command from the line below,
    # system will show you the same error (-1144), but
    # the program execution will be stopped
-- WHENEVER ERROR STOP
    CALL fgl_window_open(w_name, at_line, at_column,
                         with_rows, with_columns, true)
    LET v_status = status
    IF v_status = -1144 THEN -- the error of the wrong window
        -- cordinates

        LET error_msg = "The status = ",v_status USING "-<<<&",". \n",
                     ERR_GET(v_status)
        CALL fgl_wimmessage("Wrong Coordinates",error_msg||
                           "\nSpecify the coordinates within the screen.", "exclamation")

    ELSE
        DISPLAY "The window ", w_name, " was opened" AT 2,2
        CALL fgl_getkey()
    END IF
    CALL fgl_window_close(w_name)
CONTINUE INPUT

END INPUT
CALL fgl_window_close("w_qry_clients")

WHENEVER ERROR STOP

COMMAND "SQLCA.SQLCODE"
    "Using the SQLCA.CODE variable for analyzing the SQL statements execution"

    CLEAR SCREEN
    # We use the valid UPDATE statement which does not produce an error
    UPDATE clients SET end_date = "12/31/9999" WHERE id_client = 1
    LET v_status = status LET error_code = SQLCA.SQLCODE
    DISPLAY "A valid UPDATE statement." AT 4,2
    DISPLAY "    SQLCA.SQLCODE      = ",error_code,           ", "status = ",
            v_status AT 5,2 ATTRIBUTE(YELLOW,BOLD)

    # You can remove the comment from the WHENEVER statement below
    # to make the program terminate, if the row to be retrieved is not found
    # We do not specify the WHENEVER statement for the NOT FOUND event,
    # because the default handling is WHENEVER NOT FOUND CONTINUE

    -- WHENEVER NOT FOUND STOP

    # If we try to retrieve a non-existing row ( with id_client = 0 )
    # variable SQLCA.SQLCODE will have value NOTFOUND (=100)
    SELECT c.* INTO r_clients.* FROM clients c WHERE c.id_client = 0
    LET v_status = status LET error_code = SQLCA.SQLCODE
    DISPLAY "A SELECT statement retrieving a non-existent row." AT 6,2
    DISPLAY "    SQLCA.SQLCODE      = ",error_code,           ", "status = ",
```



```
v_status AT 7,2 ATTRIBUTE(YELLOW,BOLD)

# If we try to FETCH a row outside the active set
# variable SQLCA.SQLCODE will have value NOTFOUND (=100)
OPEN c_sel_clients
WHILE TRUE
    FETCH c_sel_clients INTO r_clients.*
    IF SQLCA.SQLCODE = NOTFOUND THEN
        LET v_status = status LET error_code = SQLCA.SQLCODE
        DISPLAY "The FETCH statement fetches a row beyond the active set."
        AT 8,2
        DISPLAY "SQLCA.SQLCODE = ",error_code,"      ","status = ",
        v_status AT 9,2 ATTRIBUTE(YELLOW,BOLD)
        EXIT WHILE
    END IF
END WHILE
CLOSE c_sel_clients

WHENEVER ERROR CONTINUE -- Comment this statement, recompile and run the
-- program to see how it will react.

# When the program tries executing UPDATE with the incorrect
# column name "id" in the WHERE clause, variable SQLCA.SQLCODE receives
# the negative value of the error code
UPDATE clients SET end_date = "12/31/9999" WHERE id = 1
LET v_status = status LET error_code = SQLCA.SQLCODE
DISPLAY "The UPDATE statement updates a non-existent column." AT 10,2
DISPLAY "SQLCA.SQLCODE = ",error_code,"      ",
        "status = ",v_status
        AT 11,2 ATTRIBUTE(YELLOW,BOLD)
LET error_msg = "status = ",v_status USING "-<<&",".",
ERR_GET(v_status)
LET error_msg = error_msg[1,70]
DISPLAY error_msg AT 12,1 ATTRIBUTE (RED,BOLD)

COMMAND "SQLCA.SQLERRD"
"Using the SQLCA.SQLWERRD variable for tracking changes made in the database"
CLEAR SCREEN
# After the successful insert into the table variable SQLCA.SQLERRD[2] will
# get the value of clients.id_client column which is automatically generated.
INSERT INTO clients VALUES(0, "804", "Alice", "Cooper", "01/01/1970", "male",
                           "ID", "10/01/1998", "10/01/2018", "Address",
                           "City", NULL, "N/A", TODAY, "12/31/9999")
LET error_code = SQLCA.SQLERRD[2]
DISPLAY "After inserting a row SQLCA.SQLERRD[2] contains the number of the "
AT 4,2
DISPLAY "inserted row which in our case is the same as in the SERIAL column."
AT 5,2
DISPLAY "SQLCA.SQLERRD[2] = ",error_code AT 6,2
ATTRIBUTE(YELLOW,BOLD)

# After the successful insert variable SQLCA.SQLERRD[3]
# will get the integer reflecting the number of the inserted rows
INSERT INTO clients VALUES(0, "840", "John", "Pizza", "10/15/1985", "male",
                           "PS", "12/01/2000", "12/01/2020", "USA",
```



```
"New York", NULL, "N/A", TODAY, "12/31/9999")
LET error_code = SQLCA.SQLERRD[3]
DISPLAY "Now SQLCA.SQLERRD[3] contains the number of inserted rows."
AT 7,2
DISPLAY "    SQLCA.SQLERRD[3] = ",error_code AT 8,2
ATTRIBUTE(YELLOW,BOLD)

# After the successful update variable SQLCA.SQLERRD[3]
# will get the number of the updated records
UPDATE clients SET end_date = "12/31/9999" WHERE id_client IN (2,5,7)
LET error_code = SQLCA.SQLERRD[3]
DISPLAY "Now SQLCA.SQLERRD[3] contains the number of updated rows."
AT 9,2
DISPLAY "    SQLCA.SQLERRD[3] = ",error_code AT 10,2
ATTRIBUTE(YELLOW,BOLD)

# After the successful deleting variable SQLCA.SQLERRD[3]
# will get the number of the deleted rows
DELETE FROM clients WHERE clients.id_client > 10
LET error_code = SQLCA.SQLERRD[3]
DISPLAY "After deleting SQLCA.SQLERRD[3] gets the number of deleted rows."
AT 11,2
DISPLAY "    SQLCA.SQLERRD[3] = ",error_code AT 12,2 ATTRIBUTE(YELLOW,BOLD)

DELETE FROM clients WHERE id_client > 10

COMMAND "SQLCA.SQLWARN"
"Using the SQLCA.SQLWARN variable for analyzing the SQL warnings"
CLEAR SCREEN

# If during the execution of any SQL statement a warning is generated
# the program will be terminated due to the WHENEVER WARNING STOP
# The default reaction is to continue the program execution

-- WHENEVER WARNING STOP

# We use the SELECT statement to assign the value of column first_name
# to record member r_clients.first_name which are of the same size.
# Nothing will be truncated and variables SQLCA.SQLAWARN[2] and
# SQLCA.SQLAWARN[1] will not get value "W"
SELECT first_name INTO r_clients.first_name FROM clients
WHERE id_client = 1
DISPLAY "Assigning values without truncating and converting." AT 4,2
LET v_sqlawarn = SQLCA.SQLAWARN[2]
DISPLAY "    SQLCA.SQLAWARN[2] = ",v_sqlawarn AT 5,2
ATTRIBUTE(YELLOW,BOLD)
LET v_sqlawarn = SQLCA.SQLAWARN[1]
DISPLAY "    SQLCA.SQLAWARN[1] = ",v_sqlawarn AT 5,30
ATTRIBUTE(YELLOW,BOLD)

# Now we assign the value of add_data with size 512 byte
# to record element r_clients.first_name which size is 20 byte
# The value will be truncated any only the first 20 bytes will be assigned.
# As the result SQLCA.SQLAWARN[2] will get "W".
# SQLCA.SQLAWARN[1] will also get "W", as at least one element of SQLAWARN
```



```

# is already set to "W".
SELECT add_data INTO r_clients.first_name FROM clients WHERE id_client = 1
    DISPLAY "Assigning values with truncating." AT 6,2
    LET v_sqlawarn = SQLCA.SQLAWARN[2]
    DISPLAY " SQLCA.SQLAWARN[2] = ",v_sqlawarn AT 7,2
    ATTRIBUTE(YELLOW,BOLD)
    LET v_sqlawarn = SQLCA.SQLAWARN[1]
    DISPLAY " SQLCA.SQLAWARN[1] = ",v_sqlawarn AT 7,30
    ATTRIBUTE(YELLOW,BOLD)

# We insert values one of which is NULL to the table.
INSERT INTO contracts VALUES(0, 1, 1, "840", "01", "26302990001", 1000.00,
                               NULL, TODAY, "12/31/9999")
# The aggregate function AVG(interest_rate) will try to include the NULL
# value into calculations which will cause SQLCA.SQLAWARN[3] to receive "W"
# value.
    SELECT AVG(interest_rate) INTO r_contracts.interest_rate FROM contracts
    DISPLAY "NULL values in aggregate functions." AT 8,2
    LET v_sqlawarn = SQLCA.SQLAWARN[3]
    DISPLAY " SQLCA.SQLAWARN[3] = ",v_sqlawarn AT 9,2
    ATTRIBUTE(YELLOW,BOLD)
    LET v_sqlawarn = SQLCA.SQLAWARN[1]
    DISPLAY " SQLCA.SQLAWARN[1] = ",v_sqlawarn AT 9,30
    ATTRIBUTE(YELLOW,BOLD)

# If the number of elements in the select list does not match
# the number of variables in the INTO clause,
# SQLCA.SQLAWARN[4] will get "W" value. * in the select list
# returns 15 values according to the number of columns in the table
# but there are only 3 variables in the INTO clause
    SELECT * INTO
r_clients.id_client,r_clients.id_country,r_clients.first_name
    FROM clients WHERE id_client = 1
DISPLAY "The selected values and target variables do not match in number."
AT 10,2
    LET v_sqlawarn = SQLCA.SQLAWARN[4]
    DISPLAY " SQLCA.SQLAWARN[4] = ",v_sqlawarn AT 11,2
ATTRIBUTE(YELLOW,BOLD)
    LET v_sqlawarn = SQLCA.SQLAWARN[1]
    DISPLAY " SQLCA.SQLAWARN[1] = ",v_sqlawarn AT 11,30
ATTRIBUTE(YELLOW,BOLD)

    DELETE FROM contracts WHERE id_contract > 6

COMMAND "Exit"
    EXIT PROGRAM
END MENU

END MAIN
#####
# This function is used as an error handler
#####
FUNCTION my_error_handler()
    DEFINE
        error_code INTEGER,

```



```
error_msg  CHAR(80)

LET error_code = status
LET error_msg = " status = ",error_code USING "-<<<& , . .",
ERR_GET(error_code)
LET error_msg = error_msg[1,40]
DISPLAY error_msg CLIPPED AT 10,1 ATTRIBUTE (RED,BOLD)

LET var_date = "12/31/2011" -- we assign the correct date and display it
DISPLAY " We assign a valid value after an error occurred: ", var_date,"."
AT 11,1

END FUNCTION
```

The Form File

The form file is used to get the window coordinates from the user. The form name is "err_window.per"

```
DATABASE FORMONLY
SCREEN SIZE 24 BY 80
{
    To invoke an error specify AT coordinates outside the 4GL screen
\gp-----q\g
\g|\g      Name:\g[f01] ]
|\g
\g|\g      AT line:\g[f02] ]
|\g
\g|\g      AT column:\g[f03] ]
|\g
\g|\g      WITH rows:\g[f04] ]
|\g
\g|\g      WITH columns:\g[f05] ]
|\g
\g|----- ] [b001 ] [b002 ] |\
\g-----d\g
\gb-----
```

}

ATTRIBUTES

```
-- all the fields are required, because they are essential
-- for successful window opening
f01 = FORMONLY.w_name, REQUIRED;
f02 = FORMONLY.at_line TYPE SMALLINT, REQUIRED;
f03 = FORMONLY.at_column TYPE SMALLINT, REQUIRED;
f04 = FORMONLY.with_rows TYPE SMALLINT, REQUIRED;
f05 = FORMONLY.with_columns TYPE SMALLINT, REQUIRED;
b001= FORMONLY.open_w, widget = "button", config="ACCEPT {Open Window}";
b002= FORMONLY.quit_but, widget = "button", config="F10 {Quit}";
```

INSTRUCTIONS

```
DELIMITERS "[ ]"
```



Index

Built-In Variables

sqlcode..... 756, 765
sqlerrd..... 765
sqlwarn 765
status..... 324, 450, 534, 754

Commenting..... 6, 95, 97, 115, 623

Data Types

ARRAY..... 244
BIGINT 18
BOOLEAN (BOOL)..... 134
BYTE 325
CHAR (CHARACTER)..... 9
CURSOR 553
Custom Data Types 236
DATE..... 56, 295
DATETIME 56
DEC (DECIMAL)..... 19
DYNAMIC ARRAY..... 245
FLOAT 20
FORM..... 103
INT (INTEGER) 18
INTERVAL..... 59
MONEY 20
PREPARED 622
RECORD 228
SMALLFLOAT 20
SMALLINT 18
STRING 9
TEXT 324
VARCHAR 9
WEBSERVICE 679
WINDOW..... 82

Data Types Conversion 45, 69, 135

Data Types Methods..... 82

CURSOR - close() 560, 604
CURSOR - declare() 554, 574, 594, 601
CURSOR - fetchAbsolute() 596
CURSOR - fetchFirst() 596
CURSOR - fetchLast() 596
CURSOR - fetchNext() 595
CURSOR - fetchPrevious() 596
CURSOR - fetchRelative() 596
CURSOR - flush()..... 605
CURSOR - free()..... 560
CURSOR - getName() 559
CURSOR - open() 556
CURSOR - put()..... 603

DYNAMIC ARRAY - append() 267
DYNAMIC ARRAY - clear() 267
DYNAMIC ARRAY - delete() 267
DYNAMIC ARRAY - getSize() 267
DYNAMIC ARRAY - insert() 267
DYNAMIC ARRAY - resize() 267
FORM - close() 107
FORM - display() 105
FORM - getHeight() 106
FORM - getWidth() 106
FORM - open() 104
PREPARED - execute() 633
PREPARED - free() 634
PREPARED - prepare() 624
PREPARED - setParameters() 629, 631
PREPARED - setResults() 629
WEBSERVICE - execute() 680
WEBSERVICE - loadMeta() 680
WEBSERVICE - selectOperation() 680
WINDOW - close() 85
WINDOW - displayAt() 84
WINDOW - open() 83
WINDOW - openWithForm() 105

Display Attributes

DISPLAY ARRAY Attributes 258
DISPLAY Attributes 11
Form Attributes 99
Form Widget Attributes 210
INPUT ARRAY Attributes 401
PROMPT Attributes 179
Window Attributes 77

Form Attributes

AUTONEXT 291
CENTER 218
CENTURY 291
COMMENT 291
DEFAULT 293
DISPLAY LIKE 538
FORMAT 294
INVISIBLE 295
LEFT 218
NOENTRY 296
PICTURE 296
PROGRAM 328
REQUIRED 297
RIGHT 218
VALIDATE LIKE 538



VERIFY.....	538	fgl_dialog_getkeylabel().....	435
WORDWRAP	209	fgl_dialog_infield().....	340
Form File Objects		fgl_dialog_setbuffer()	343
Browser.....	214	fgl_dialog_setcurrline()	412
Button.....	262	fgl_dialog_setcursor()	371
Calendar.....	302	fgl_drawbox()	102
Checkbox.....	367	fgl_drawline()	101
Combo Box	381	fgl_driver_error()	774
Dynamic Label	213	fgl_fglgui().....	50
Form Tabs	297	fgl_file_dialog()	218
Function Field	348	fgl_find_table()	477
Hotlink	304	fgl_form_close()	103
Image	216	fgl_form_display()	99
Radio Button.....	364	fgl_form_open()	99
Screen Array	251, 408	fgl_formfield_getoption()	371
Screen Grid.....	255	fgl_getenv()	63
Screen Record.....	231, 537	fgl_gethelp()	324
Static Object	96	fgl_getkey()	157, 345
Text Field	97	fgl_getkeylabel()	435
Text Filed	205	fgl_getuitype()	50
Form File Sections		fgl_getwin_height()	85
ACTIONS	428	fgl_getwin_width()	85
ATTRIBUTES.....	206, 536	fgl_getwin_x()	87
DATABASE.....	94, 535	fgl_getwin_y()	87
INSTRUCTIONS.....	207	fgl_keydivider()	437
KEYS	428	fgl_keyname()	347
SCREEN.....	95, 205, 299	fgl_keyqueue()	347
TABLES	536	fgl_keyval()	290, 346
Functions		fgl_lastkey()	290, 346
arg_val()	49	fgl_list_clear()	383
arr_count()	411	fgl_list_count()	385
arr_curr()	410	fgl_list_find()	385
ASCII	47	fgl_list_get()	386
avg()	597, 740	fgl_list_insert()	387
count()	598, 741	fgl_list_restore()	391
create_menu()	452	fgl_list_set()	388
cursor_name()	559	fgl_list_sort()	390
downshift()	47	fgl_message_box()	144
err_get()	766	fgl_native_code()	772
err_print()	766	fgl_native_error()	772
err_quit()	766	fgl_putenv()	63
errorlog()	770	fgl_random()	48
exec_program()	455	fgl_scr_size()	266, 411
extend()	65	fgl_setactionlabel()	435
fgl_bell()	771	fgl_setkeylabel()	192, 435
fgl_bufertouched()	344	fgl_setsize()	88
fgl_column_info()	477	fgl_settitle()	89
fgl_dialog_getactionlabel()	435	fgl_winbutton()	146
fgl_dialog_getbuffer()	343	fgl_window_clear()	81
fgl_dialog_getcursor()	369	fgl_window_close()	81, 103
fgl_dialog_getfieldname()	370	fgl_window_current()	80



fgl_window_getoption()	267
fgl_window_open()	76
fgl_winmessage()	143
fgl_winquestion()	147
fgl_winsize()	86
field_touched()	338
get_fldbuf	341
infield()	339
length()	48
max()	597, 741
menu_add_option()	452
menu_add_submenu()	453
menu_publish()	454
min()	597, 741
num_args()	50
ORD()	46
percent()	741
scr_line()	412
set_menu_server()	451
showhelp()	321
sizeof()	266
startlog()	768
sum()	597, 740
trim()	48
upshift()	47
winexec()	448
winexecwait()	448
winshellexec()	449
winshellexecwait()	449
Operators	
Arithmetic (+, -, *, /, ** MOD)	20, 68
ASCII	735
Boolean	133
CLIPPED	12
COLUMN	13, 735
Comma (,)	14
CURRENT	64
DATE	66
DAY	67
Double Pipe ()	14
LIKE	140
LINENO	737
MATCHES	140
MONTH	67
PAGENO	737
Relational	132
SPACE	13, 736
TIME	66
TODAY	64
UNITS	64
USING	22, 68
WEEKDAY	67
WORDWRAP	738
YEAR	67
Script Files 114	
border	372
color	121, 254
default	118
deltaX	124
deltaY	124
fgloverwrite	373
font	119
hidden	117
MDImode	451
Named Strings	117
no-form	117
Replacing strings	118
templates	126
titlebar	160
tooltips	436
Wildcard Symbols	116
windowStyle	125
Statements	
ALTER TABLE	515
BEGIN WORK	570
CALL	39, 44, 669, 760
CASE	137
CLOSE	560
COMMIT	571
CONSTRUCT	647
CREATE SYNONYM	532
CREATE TABLE	469
CREATE TEMP TABLE	475, 569
CREATE VIEW	530
CURRENT WINDOW IS	79
DATABASE	467, 572
DECLARE	552, 574, 594, 600
DEFER	159
DEFINE	8
DEFINE... LIKE	532
DELETE	519
DELETE SYNONYM	532
DELETE VIEW	531
DISPLAY	6
DISPLAY ARRAY	252
DISPLAY AT	10
DISPLAY BY NAME	235
DISPLAY FORM	98
DISPLAY TO	209, 233
ERROR	290, 320
EXECUTE	631
EXIT PROGRAM	159, 450



EXIT REPORT.....	742	OPEN FORM.....	98
FETCH.....	556, 595	OPEN WINDOW	75
FINISH REPORT	703	OPEN WINDOW ... WITH FORM.....	100
FLUSH.....	604	OPTIONS 100, 159, 183, 193, 292, 500, 605	
FOR	164	OPTIONS.....	336
FOREACH	592	OUTPUT TO REPORT	702
FREE	560, 634	PAUSE.....	742
FUNCTION.....	30, 38, 669	PREPARE.....	620
GLOBALS.....	28	PRINT	733, 739
GOTO.....	157	PROMPT	156, 176
GRANT	475	PUT.....	603, 630
IF135		REPORT.....	694, 708
INITIALIZE	235, 534	RETURN	43
INPUT	283	ROLLBACK WORK.....	571
INPUT ARRAY	399	SELECT	491, 553, 597
INPUT BY NAME.....	335	SKIP.....	743
INSERT	517, 600	SLEEP	90, 156
LABEL	157	START REPORT	697
LET	10	TERMINATE REPORT	703
LOAD	488	TRY...CATCH.....	761
LOCATE.....	326	UNLOAD	498
LOCK TABLE	572	UNLOCK TABLE	573
MAIN.....	5	UPDATE.....	520
MENU.....	184	VALIDATE.....	533
MESSAGE	319	WHENEVER.....	755
NEED	742	WHILE.....	168
OPEN	555, 628		

