
Mastering 4th Dimension

The RESOLVE POINTER command

by Walt Nelson

If you have been writing 4D code for several months or years you have probably decided that it is a good idea to write *generic code*; code you can reuse within an application—or even better—code that you can reuse in future 4D applications. 4D, in my opinion, is a wonderful tool for writing generic code. The most important 4D object that helps to make generic code possible is the Pointer. In this article, we will talk about Pointers in general, and about the command **RESOLVE POINTER** in particular. We will start out with a review of fundamental concepts about Pointers in 4D, and then gradually progress into “black-belt territory” to talk about some advanced uses of pointers and the command **RESOLVE POINTER**.

The command **RESOLVE POINTER** was first introduced in 4D Version 6. Following the release of Version 6, I have found the command **RESOLVE POINTER** to be extremely useful for writing generic re-usable code. In fact, I will make the following bold statement: *In terms of the ability to write generic re-usable code, the command RESOLVE POINTER was the most important new command introduced in 4D Version 6.*

What is a 4D pointer?

Just in case you are new to 4D and are not sure what a Pointer is or how it is used in 4D, let's start by getting an understanding of 4D Pointers.

Note 1: If you are an experienced 4D developer who thoroughly understands Pointers already, you can skip directly to the sub-heading “Why RESOLVE a POINTER?”

Note 2: Pointers in 4D are somewhat different than pointers in other languages such as C or C++. For the remainder of this article, when I speak of Pointers, I am talking strictly about pointers as they are used in 4D.

Here are some facts about 4D's Pointers:

A Pointer in 4D is a special type of variable that can be used as a way to refer to an object indirectly, instead of calling the object by its real name. A pointer “points” to an object.

A Pointer in 4D is somewhat like a computer system's Alias or Shortcut for an object, but the 4D Pointer is far more versatile. An Alias or Shortcut always refers to the same object; a 4D Pointer can refer to different objects, depending on the circumstances.

The ability to “...refer to different objects, depending on the circumstances” is what gives a 4D Pointer its power. The same pointer might refer to an alpha field in one instance, a

date variable in a second instance, and a numerical array in a third instance—all without violating the 4D Compiler's rules of data typing!

How are 4D pointers created?

There are several different ways to create a pointer to an object in 4D because several 4D commands and functions return pointers to objects. The most common way to create a pointer to an object, however, is to place an arrow symbol (->) in front of the name of the object:

```
pMyStringPtr := ->vsMyString
                creates a pointer to the variable vsMyString
pMyTblPtr := ->[TableName]
                creates a pointer to the table [MyTable]
pMyFldPtr := ->[TableName]Field
                creates a pointer to the field [MyTable]Field
pMyArrayPtr := ->aMyArray
                creates a pointer to the array aMyArray
```

To find out the data that a Pointer is referring to, you *de-reference* the pointer—you place an arrow symbol behind the name of the pointer:

```
pMyStringPtr->
                returns the string "MyExample"
pMyTblPtr->
                returns the table [MyTable]
pMyFldPtr->
                returns the field [MyTable]MyField
pMyArrayPtr->
                returns the array aMyArray
```

Fields and arrays: Dual-syntax

When you begin using 4D Pointers, you may have trouble understanding how to use pointers to fields and arrays because of the dual-syntax of field names and array names. For example:

[MyTable]Field can mean the general “class” of data that is stored in the field [MyTable]Field; however, it can also mean a specific instance of [MyTable]MyField (that is, the data in a single record).

QUERY([MyTable];[MyTable]MyField...) refers to the general class of data that is stored in the field.

[MyTable]MyField := "Smith" refers to a specific instance of the data.

aMyArray can mean the general class of data that is stored in the array aMyArray; however, it can also mean the currently selected element (row) of the array aMyArray.

COPY ARRAY (aMyArray...) refers to the general class of data that is being stored in the array.

aMyArray:=3 refers to a specific row of the array. It is saying: "Make the third row of the array the current row."

Understanding the double-duty of field names and array names is difficult enough when you are referring to those objects directly by their names; it is even more difficult when you are referring to those objects with the use of Pointers!

The general rule to remember is this: *In every case where you can use the object name, you can substitute the de-referenced pointer name.* For example:

```
QUERY (pMyTablePtr->;pMyFieldPtr->...)
```

is the same as:

```
QUERY ([MyTable];[MyTable]MyField...)
```

and

```
pMyFieldPtr->:="Example"
```

is the same as:

```
[MyTable]MyField:="Example"
```

and

```
COPY ARRAY (pMyArrayPtr->...)
```

is the same as:

```
COPY ARRAY (aMyArray...)
```

and

```
pMyArrayPtr-> := 3
```

is the same as:

```
aMyArray := 3
```

and finally:

```
pMyArrayPtr->3 := "Example"
```

is the same as:

```
aMyArray3 := "Example"
```

A nonsense poem that might help you to remember this principle is: "Wherever you can use the sparrow, you can substitute the pointer and the arrow."

Why use pointers?

4D developers use Pointers to write generic code that can be re-used without being re-written. Generic code is a good thing for at least three reasons:

- Re-using code is faster than writing the same thing over and over again from scratch.
- Generic re-usable code is already debugged. Since debugging a routine can be more time-consuming than writing the routine, this is a significant time savings.

- Generic routines help to standardize your code. Because of the very nature of generic routines, they encourage you to use standard naming conventions and standard coding conventions. Standardized code is easier to understand, easier to de-bug, and easier to maintain.

Practical uses of pointers in 4D

The basic strategy of using pointers is:

- Write a routine that uses Pointers instead of directly using fields or variables.
- Call that routine and pass Pointers as parameters.
- Inside the routine, assign the Pointers so that they are pointing to the variables and fields you passed as parameters.

Here is an example of a routine that will get all the records in a table, sort the records in ascending order, and then show the list of records in an output form.

```
`Project Method: Show All Clients
ALL RECORDS ([Clients])
ORDER BY ([Clients];[Clients]MyField;>)
INPUT FORM ([Clients;"Input")
OUTPUT FORM ([Clients;"Output")
MODIFY SELECTION ([Clients];*)
```

Here's a generic version of this method using Pointers

```
`Project Method: ShowAllRecords
ALL RECORDS (pTablePtr->)
ORDER BY (pTablePtr->;pFieldPtr->>)
INPUT FORM (pTablePtr->;"Input")
OUTPUT FORM (pTablePtr->;"Output")
MODIFY SELECTION (pTablePtr->*)
```

Both routines will display all the records in the [Clients] table. However, the Pointer version can also be used for the [Products] table, the [Invoices] table, and so on. To make this work, all we have to do is add some definitions at the beginning of our generic method, *ShowAllRecords*.

```
`Project method: ShowAllRecords
C_POINTER ($1;pTablePtr;$2;pFieldPtr)
pTablePtr := $1
pFieldPtr := $2
ALL RECORDS (pTablePtr->)
ORDER BY (pTablePtr->;pFieldPtr->>)
INPUT FORM (pTablePtr->;"Input")
OUTPUT FORM (pTablePtr->;"Output")
MODIFY SELECTION (pTablePtr->*)
```

When you want to display all the records in any table in the database, you can do it by calling *ShowAllRecords* and passing two pointers as parameters: a pointer to the table, and a pointer to the field that you want to sort by.

```
ShowAllRecords (->[Clients];->[Clients]Company_Name)
ShowAllRecords (->[Invoices];->[Invoices]Invoice_Number)
ShowAllRecords (->[Products];->[Products]Description)
```

Here are just a few of the other instances in which you can use pointers to save time and standardize code:

- Generic **Query** routines that can be re-used to search in any table.
- Generic **Add Record** routines that can be re-used to add a record to any table.
- Generic **Date** routines that can be re-used to manipulate any date or series of dates.
- Generic **String** routines that can be re-used to manipulate any string or series of strings.
- Generic **Array** routines that can be re-used to manipulate any array or series of arrays.
- Generic **Drag-and-Drop** routines that be re-used to drag and drop between objects.

Now you have a solid conceptual understanding of 4D Pointers. You know enough to begin using Pointers to help you write generic code that you can re-use over and over. Start with some examples written by experienced developers, make sure you understand how those examples work, and then start adapting those examples for your own use.

Why RESOLVE a POINTER?

Now we can discuss the main topic of this article: the command **RESOLVE POINTER**, a new command that was introduced in 4D Version 6. Even though Pointers were extremely flexible before Version 6, there were still certain situations in which we did not have all the tools we needed in order to write generic code.

You know that to **de-reference** a pointer is to find out the **data** that is contained in the object that the pointer is referring to. The ability to de-reference a pointer has always been very handy; however, for the purpose of writing certain generic routines, we sometimes needed to know the **name of the object** to which the pointer was referring. If the object was a table or a field, the 4D language provided a way to find out its name; however, if the object was a variable or an array, there was no way—prior to Version 6—to find out the name of the variable to which the pointer was referring.

Why would you want to know the name of the object that a pointer was referring to? Let's consider an example.

Assume you have a 4D-to-SQL application in which the data is stored in a back-end SQL database.

When you want to present data to the user, you do a query of the SQL back-end and retrieve the desired columns in the form of 4D arrays. You display these arrays in an “output form.”

To view or modify a single record, the user double-clicks a row of the array. You sense the double-click event, do a query of the SQL back-end, and copy the desired fields into 4D variables. You then present those variables to the user in a 4D input form or dialog.

Suppose the name of one of your SQL tables is “Clients.” You set up your naming conventions as follows:

SQL column: Clients.Company_Name

4D array: aClients_Company_Name
 4D variable: vClients_Company_Name

SQL column: Clients.Address1
 4D array: aClients_Address1
 4D variable: vClients_Address1

SQL column: Clients.Telephone_No
 4D array: aClients_Telephone_No
 4D variable: vClients_Telephone_No

You can see the naming convention here. The 4D arrays are named:

"a" + TableName + "_" + FieldName

and the 4D variables are named:

"v" + TableName + "_" + FieldName

- If you know the SQL table name and column name, you can figure out the array name and the variable name.
- If you know the variable name, you can figure out the array name, the SQL table name, and the SQL column name.
- If you know the array name, you can figure out the SQL table name, the SQL column name, and the variable name.

As human beings, we can easily see those patterns. But how do we teach 4D to figure out those names, and how can we use these naming conventions to help us write generic code? The answer is the 4D command **RESOLVE POINTER**. *To RESOLVE a POINTER is to find out the name and/or the number of the object that the pointer is referring to.*

Here is the syntax of the **RESOLVE POINTER** command:

RESOLVE POINTER (pPointer; sObjName; iTableNr; iFieldNr)

To get a valid result from this command, the pointer pPointer must already exist. You should also declare the variables sObjName, iTableNr, and iFieldNr:

C_STRING(31;sObjName)
C_LONGINT(iTableNr;iFieldNr)

Now let's see what **RESOLVE POINTER** can tell us. Suppose we create a pointer named pObjectPtr and we reference various objects as follows:

pObjectPtr := -> vClients_Company_Name
 pObjectPtr := -> aClients_Address1
 pObjectPtr := -> aClients_Zip3

Suppose we call **RESOLVE POINTER** in each of the above cases.

RESOLVE POINTER (pObjectPtr; sVarName; iTableNr;iFieldNr)

Here are some examples of the value of the results of the **RESOLVE POINTER** command:

If pObjPointer is	sVarName will return
->vClients Company Name	"vClients Company Name"
->aClients Address1	"aClients Address1"
->aClients Zip{3}	"aClients Zip"

The third example is and array element. In this case **RESOLVE POINTER** will return the array element number in the variable `iTableNr`.

How can we use this information to help us write generic code? Well, suppose we want to write a routine that would function as follows:

- If we pass a pointer to a 4D array, the generic routine will derive the name of the matching SQL field; derive the name of the matching 4D variable; create a pointer to the 4D variable; and copy the current value of the current array element into the 4D variable.
- If we pass a pointer to a variable, the generic routine will derive the name of the matching SQL field; derive the name of the matching array; create a pointer to the array; and copy the value of the variable into the current array element.

If you have ever written a 4D-to-SQL connection, you know that one of the biggest time-wasters is the “little” mistake: you make a simple typing error in a SQL column name, a 4D variable name, or a 4D array name; then you execute the code and get SQL errors; and you waste several minutes (or several hours!) tracking down the mistake. If you are using the routine `SQL_Names_Pointers_Derive` with the naming conventions that we recommend, you will reduce typographical errors in field, variable, and SQL names to almost zero.

On this issue’s examples disk, you will find these three example routines:

`SQL_Names_Pointers_Derive` is a project method that receives two parameters: a Longint and a pointer. This method assumes that you are working with a SQL back-end database, and that you are using the naming conventions that we mentioned above (“v” prefix for variables, “a” prefix for arrays, followed by the SQL table name, followed by an underscore, followed by the SQL column name). Here are some examples of how you would call this routine:

```
SQL_Names_Pointers_Derive(1;->Clients_Company_Name)
SQL_Names_Pointers_Derive(1;->aClients_Company_Name)
```

The method returns to you the following variables and pointers:

sSQLTableName	The name of the SQL table
sSQLColumnName	The name of the SQL column
pVarPtr	A pointer to the variable associated with this SQL column
sVarName	The name of the variable as a string
pVarPtr->	The data that is contained in the variable
pArrayPtr	A pointer to the array associated with this SQL column
sArrayName	The array name as a string
pArrayPtr->Element	The data that is contained in a specific array element

When you are writing your SQL code, use this method to eliminate the mistakes you might make when entering variable names, array names, SQL table names, and SQL column names. If you can just type one parameter correctly, all the corresponding names will be automatically generated for you.

Note: Remember that this method assumes that all variable names and array names have already been declared in compiler directives. One of the unbreakable rules of 4D is that you cannot create a pointer to an object that doesn’t already exist.

The method `Test_Derive` allows you to see the methods `SQL_Names_Pointers_Derive` & `SQL_Names_Pointer_Loop` in action. Paste the three methods into a 4D structure, go to the User environment, and Execute the method `Test_Derive`. If you want to see step-by-step as it derives the pointers, you can turn on the TRACE statements that are in the method `Test_Derive`.

The method `SQL_Names_Pointers_Loop` is a time-saving device. When you need to copy data from variables to array elements, or from array elements to variables, you pass a stream of several variables or arrays to the method `SQL_Names_Pointers_Loop`. This method loops through the list of pointers and then calls the method `SQL_Names_Pointers_Derive` for each passed pointer.

A note about speed

You may think that all these lines of code will slow things down; however, that is not the case. Even when uncompiled, on a 300mhz or faster machine, 4D can process five variables or arrays faster than you can blink your eyes. In compiled mode, on a 200mhz or faster machine, 4D can process 25 variables or arrays in the method `SQL_Names_Pointers_Loop` faster than you can blink your eyes. There is very little performance penalty for using this method, but the benefits are significant: you reduce the time wasted debugging keyboard errors—variable names, array names, and SQL column names.

Summary

The command **RESOLVE POINTER**, which was newly introduced in 4D Version 6, gives you a tool to take generic programming to the “next level.” Because you are able to find out the name of any object that is referred to by a pointer, you can generate pointers to *associated* objects that have been named based on a set of standard naming conventions. You can also copy data between data objects: fields, variables, and array elements.

This is a “black-belt” 4D concept. Until you are very familiar with this style of programming, it will take you quite a while to write generic routines using **RESOLVE POINTER**. However, once you have the routine written and debugged, you can re-use it over, and over, and over.

Source

If you don’t get the Dimensions examples disk here is the source for the three methods along with examples of how you can use them:

```
`Method: SQL_Names_Pointers_Derive
`Assumes naming convention "v@" for variables, "a@" for
`arrays.
```

```

`Copyright 1999 by Dimensions Magazine & Walt Nelson

`Declare the incoming parameters.
C_LONGINT ($1;$iElement)` The array element (if applicable)
C_POINTER ($2;$pObjectPtr)` The name of the 4D object

C_LONGINT ($iPosition)` A Longint variable that we will need.

`Declare standard pointers & variable names that we will use
`over and over.
C_POINTER (pVarPtr;pArrayPtr)
C_STRING (31;sObjectName;sVarName;sArrayName)
C_STRING (31;sSQLTableName;sSQLColumnName)

`Read the two passed parameters.
$iElement:=$1
$pObjectPtr:=$2

RESOLVE POINTER ($pObjectPtr;$sObject-
Name;$iTableNr;$iFieldNr)

Case of
: ($sObjectName="v@") If we passed a variable...
pVarPtr:=$pObjectPtr
`Copy the object pointer into our standard var pointer.
sVarName:=$sObjectName
`Copy the object name into our standard VarName variable.
sArrayName:="a"+Substring (sVarName;2)
`Derive the array name.
pArrayPtr:=Get pointer (sArrayName
`Derive a pointer to the array name.

If ($iElement>0)` If we passed an array element number...
pArrayPtr->$iElement:=pVarPtr->
`Copy the data from the variable into the array element.
End if

Else `If We passed an array...
pArrayPtr:=$pObjectPtr
`Copy the object pointer into our standard Array pointer.
sArrayName:=$sObjectName
`Copy the object name into our standard Array Name
`variable.
sVarName:="v"+Substring (sArrayName;2)
`Derive the variable name
pVarPtr:=Get pointer (sVarName)
`Derive a pointer to the variable

If ($iElement>0)` If we passed an array element number
pVarPtr->:=pArrayPtr->$iElement
`Copy the data from the array element into the variable
End if

End case

`Derive the SQL table name and column name...
$iPosition:=Position ("_",sObjectName)
`Find the first underscore character
sSQLTableName:= Substring ($sObjectName;2;
($iPosition-2))` Derive the SQL table name
sSQLColumnName:=Substring ($sObjectName;
($iPosition+1))` Derive the SQL column name

```

```

`*****End of method*****

`Method: SQL_Names_Pointers_Loop
`Uses Parameter Indirection $$ to loop through passed
`parameters.

C_LONGINT ($1;$iElement;$iCounter)
$iElement:= $1
$iParameters := Count parameters
For ($iCounter;2;$iParameters)
`Start the loop at the 2nd parameter
SQL_Names_Pointers_Derive($iElement;$iCounter)
End for

`*****End method*****

`Method: Test_Derive
`Tests the method SQL_Names_Pointers_Derive
`Copyright 1999 by Dimensions Magazine & Walt Nelson

C_STRING (80;vClients_Company_Name)
C_TEXT ($tDerived)
ARRAY STRING (80;aClients_Company_Name;1)

`Test passing a variable pointer...
`TRACE
vClients_Company_Name:="Walt Nelson & Associates"
aClients_Company_Name1:=""
SQL_Names_Pointers_Derive(->vClients_Company_Name;1)

$tDerived:="Variable name is: "+sVarName+ Char(13)
$tDerived:=$tDerived+"Array name is: "+
sArrayName+Char(13)
$tDerived:=$tDerived+"New Array value is: "+
aClients_Company_Name1+Char(13)
$tDerived:=$tDerived+"SQL Table name is: "+
sSQLTableName+Char(13)
$tDerived:=$tDerived+"SQL Column name is: "+
sSQLColumnName+"."+sSQLColumnName+Char(13)

ALERT ($tDerived)

`Now test passing an array...

`TRACE
aClients_Company_Name1:="Dimensions Magazine"
vClients_Company_Name:=""

SQL_Names_Pointers_Derive (->aClients_Company_Name;1)

$tDerived:="Variable name is: "+sVarName+ Char(13)
$tDerived:=$tDerived+"Array name is: "+
sArrayName+Char(13)
$tDerived:=$tDerived+"New variable value is: "+
vClients_Company_Name+Char(13)
$tDerived:=$tDerived+"SQL Table name is: "+
sSQLTableName+Char(13)
$tDerived:=$tDerived+"SQL Column name is: "+
sSQLColumnName+"."+sSQLColumnName+Char(13)

ALERT($tDerived)

```

```

`-----
`Test passing multiple variables...

`Declare the variables and fill them with text...
C_STRING (31;vClients_Company_Name)
C_STRING (65;vClients_City)
C_STRING (11;vClients_Zip)
vClients_Company_Name:="Walt Nelson & Associates"
vClients_City:="Cupertino"
vClients_Zip:="95014"

`Declare the arrays and make sure the elements are empty...
ARRAY STRING (31;aClients_Company_Name;1)
ARRAY STRING (65;aClients_City;1)
ARRAY STRING (11;aClients_Zip;1)
aClients_Company_Name1:=""
aClients_City1:=""
aClients_Zip1:=""

`Call the Loop
SQL_Names_Pointers_Loop (1;->vClients_Company_Name;-
->vClients_City;->vClients_Zip)

```

```

$tDerived:="aClients_Company_Name1 is: "+
aClients_Company_Name1+Char(13)
$tDerived:=$tDerived+"aClients_City1 is: "+
aClients_City1+Char(13)
$tDerived:=$tDerived+"aClients_Zip1 is: "+aClients_Zip1

ALERT ($tDerived)

`*****End of method*****

```

About the author

Walt Nelson is a writer, trainer, motivational speaker, and 4D developer/consultant. Walt has been doing 4D development since 1987; he was a participant in the original "Silver Surfer" 4D Beta Test project. From his home base in the San Francisco Bay Area, walt manages his consulting, publishing, and 4D Mentoring business. Walt is the author of three books on 4th Dimension, all of which can be ordered at his web site, www.WaltNelson.com.