# Pointers

## By David Adams and Dan Beckett

## Introduction

Pointers are one of the most powerful—and potentially most confusing—features of the 4D language. You do not have to use pointers to program 4D, but using them allows you to create much more elegant solutions and code that can be resused more easily. Also, to avoid using global variables to communicate data to and from custom methods, pointers are essential. This article examines the syntax, benefits, and limitations of coding with pointers.

## About Pointers

### Pointers Are References

Pointers are *references* to objects. You can use a pointer to an object in place of the object itself anywhere in the 4D language. Here is a simple example of a pointer in use.

```
C_POINTER(pMyPointer) `pMyPointer is a pointer.
C_STRING(255;sMyString) ` sMyString is a string.

sMyString:="Hello world!"
pMyPointer:=->sMyString ` pMyPointer becomes a pointer to sMyString.

   ` Because pMyPointer points to sMyString, it can be used in place of sMyString.
ALERT(pMyPointer->) ` Display what pMyPointer is pointing at.
```

Pointers allow you to write generic routines that can handle different tables and data of different types. This reduces the time and code needed to complete a project and produces code that is more portable. Here is an example of a pointer used in place of a direct table reference.

```
` ALL_RECORDS (Pointer)
`
` A table-dependent replacement for 4D`s ALL RECORDS command.

C_POINTER($1;$Table)

$Table:=$1

Case of
   : ($Table->=[Customers]) ` If it`s the Customers table…
         ` …only get the active customers.
      QUERY([Customers];[Customers]Archived=False)

   : ($Table=->[Invoices]) ` If it`s the Invoices table…
         ` …only get the unpaid invoices.
      QUERY([Invoices];[Invoices]Paid=False)

Else ` For any other table, do the usual ALL RECORDS.
   ALL RECORDS($Table->)
End case
```

The advantage in this simple example is that different tables can be reacted to differently in the same generic method.

## Pointers Are Programmer Data

Pointers aren't *user* data, they're *programmer* data. They are variables that you use in your code to change how the code works or what it works on. The user never sees, works with, or knows about pointers. For this reason, pointers are the only variable type for which there is no corresponding field type in 4D.

## Pointers Are Simple!

Ordinary pointer syntax is straightforward: put the reference symbol (->) in front of an object when you create a pointer, and put the dereference symbol (->) after the pointer to use it in the place of the object.

**Note** *The pointer reference and dereference symbol for 4D 2.x and 4D 3.x Macintosh is », as you may still find in many 4D reference materials and publications.*

| Creating a Pointer | Using a Pointer | Same As |
|---|---|---|
| $Table:=->[Customers] | **QUERY**($Table->) | **QUERY**([Customers]) |

This covers better than 99% of what you need to know about pointers. OK, you're done! You can go home now. If you want to know about the remaining 1%, read on.

## Why Pointers are Confusing

### You're Not Alone!

If you find pointers confusing or hard to understand, you are not alone. Pointers *are* confusing and hard to understand at first for most people. (Once you understand pointers, though, using them is simple.) There are valid reasons that pointers can be difficult to understand:

- ❖ Pointers refer to objects *indirectly*.
- ❖ The pointer reference and dereference symbols are the same.
- ❖ Pointers can point to very different objects.
- ❖ The effect of dereferencing a pointer may change depending on context.
- ❖ 4D's pointer implementation is not exactly like that of other languages.

### Pointers Refer to Objects Indirectly

The nature of pointers is that they refer to objects *indirectly*. A pointer is an address, or reference. This means that when you are working with a pointer you are always one step further away from the object than when you refer to it directly in your code. This is, by nature, harder to understand. When you work with pointers you need to keep in mind what the pointer refers to as you write your code.

You may find it easier to learn about pointers through experimentation than through reading other people's code. (At least you know what your own code is supposed to do!) One way to become familiar with pointers for the first time is to write a routine *without* pointers, then, once it is working properly, replace the literal table, field, array and variable references with pointers. If you are learning about pointers, try this approach.

### The Pointer Reference and Dereference Symbols Are the Same

The symbol for creating a pointer and the symbol for using an object referred to by a pointer are the same. This is sometimes confusing since this is the only operator in the 4D language whose position determines its function. Let's look at another simple example of the pointer symbol in action:

```
sMyVar:="Buon giorno mondo!" ` A string variable.
$pMyPointer:=->sMyVar ` $pMyPointer points at sMyVar
ALERT ($pMyPointer->) ` Displays what $pMyPointer is pointing at.
```

The pointer reference and dereference symbols are the same symbol, but are used in different positions. The *position* of the symbol determines it's function. If the symbol precedes an object, you create a pointer, as in the second line of the example. If it follows a pointer variable, you are dereferencing the pointer to get at the object the pointer references, as in the third line.

### Pointers Can Point to Several Types of Objects

One of the great features of pointers is that they can point to many objects: variables, arrays, tables, fields and array elements. On the other hand, this freedom makes code that uses pointers hard to understand at times. The same pointer variable can point to each of these different object types at different times. In order to understand a code segment that uses pointers, you need to understand exactly what the code is doing and what the pointers currently references. If you are not yet thoroughly familiar with the 4D language, this can be a bit of a challenge. Let's look at what a 4D pointer can and can't point to:

**Pointer Targets**

| Can Point To | Can't Point To |
|---|---|
| Process variables | Local variables |
| Process arrays | Local arrays |
| Interprocess variables | Lists |
| Interprocess arrays | Forms |
| Tables | Menu bars |
| Fields | Methods of any kind |
| Array elements | Processes |
| | Windows |
| | Documents |
| | Resources |
| | Plug-ins |
| | Pictures |

### You Can't Point to Locals

You can't use a pointer to a local variable or a local array. You can *store* pointers in local variables and arrays, but they cannot *point at* local variables or arrays. For example, the following code is fine:

```
C_POINTER($pCurrTable)
$pCurrTable:=Current form table
```

The next piece of code, however, is not acceptable because $pDateVar points to the local variable $dCurrent:

```
C_POINTER($pDateVar)
C_DATE($dCurrent)

$dCurrent:=Current date

$pDateVar:=->$dCurrent
```

In some programming languages you can pass a pointer to a local variable to another subroutine which can then manipulate the variable local to the first method. In 4D, pointers to local variables and local arrays are not supported features, either within a routine or as arguments to a subroutine. That's just how 4D works.

### The Result of Dereferencing a Pointer Depends on Context

4D does a great job of figuring out how to use a pointer based on where you use it. Unfortunately, this is also the biggest single reason people find 4D pointers confusing. For example, if you have a pointer to an array, you can use the pointer to populate the array, set the currently selected element number, copy the array, or access array elements as shown below:

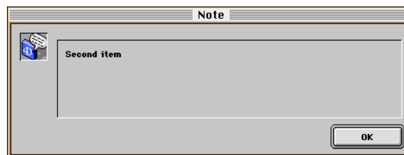
```
ARRAY STRING(20;asMyArray;2)
C_POINTER($pArray)

  ` Create a pointer to the new array:
$pArray:=->asMyArray

  ` Use the pointer to assign values to the array elements:
$pArray->{1}:="First item"
$pArray->{2}:="Second item"

  ` Use the pointer to set the second element as the current element:
$pArray->:=2

  ` Display the contents of the currently selected array element:
ALERT($pArray->{$pArray->})
```




*The second item displayed in an alert dialog.*

The pointer used in this example is very flexible, but doesn't help make the code any easier to read. If you are familiar with how 4D manages arrays, then the preceding code makes sense. If not, using pointers only obscures it further.

Consider another example. You have a pointer to a field. Does the pointer point to the field itself, or to the value of that field in the current record? It depends.

```
C_POINTER($pField)
C_STRING($sCurrPart)

  ` Create a field pointer (you can also use the 'Field' function to create
  ` a field pointer):
$pField:=->[Parts]Name

  ` Now we copy the value of the field out of the current record:
$sCurrPart:=$pField->

  ` Now we use the field references in a query:
QUERY([Parts];$pField->="Black dots")
```

The same pointer serves different purposes depending on context. This is exactly how 4D handles field references without pointers, as well. Pointers don't change any of 4D's underlying behavior but can make it harder to understand at first.

If you are having trouble with pointers in contexts like the ones we've shown, review the basics of how 4D handles these situation *without* pointers. Once you understand the underlying behavior, using pointers adds little new complexity.

## Working With Pointers

### Pointers Are Not Slow

Since a pointer is an indirect reference, it may seem that they make code that uses them slow. This is not true. We conducted timing tests to determine the impact of pointer use on execution speed and found that it makes little meaningful difference.

We compared loops with and without pointers. To exaggerate the difference we repeated the loops one million times. In an interpreted database it was about 5% slower to use pointers, and in a compiled database it was about twice as slow to use pointers (200 vs. 105 ticks). Pointers are twice as slow! Yes, but the actual difference is under two seconds in a loop of one *million* iterations. Here is a case where you could micro-optimize something by a factor of two and effectively gain nothing.

## Get Pointer Is Slow

In contrast, **Get pointer** is one of the slowest functions in the 4D language. This doesn't mean that you shouldn't use it, it simply means you should use it judiciously. For example, avoid using **Get pointer** in a loop to control a user interface on equipment where the code does not execute quickly enough.

## Get Pointer Does Not Create Variables

**Get pointer** takes a string and returns a pointer to a variable or array with that name. **Get pointer** cannot create pointers to fields or tables. (Use the **Table** and **Field** functions for this.) The following line of code populates $pMy_pointer with a pointer to the variable sMyString:

```
$pMy_pointer:=Get pointer("sMyString")
```

In order for this to work in a compiled database, sMyString must already exist. **Get pointer** does not create variables, it creates a *reference* to a variable. People often run into trouble using **Get pointer** in a **For** loop to create a series of pointers to objects that do not exist. The following code fragment populates a local pointer array called $apButtons with pointers to the variables button1…button100. Unfortunately, these buttons do not exist.

```
C_LONGINT($i)

ARRAY POINTER($apButtons;100)

For ($i;1;100)
   $apButtons{$i}:=Get pointer("button"+String($i))
End for
```

In a compiled database this code executes without generating an error. If, however, you attempt to use any of these pointers you immediately get an error. Why? Because the variables to which the code is supposed to create pointers do not exist. You must explicitly declare all variables and arrays that you use in a compiled database.

# Nil

4D's **Nil** function is an easy way to test if a pointer is valid before using it. **Nil** is a Boolean function that tells you if a pointer points at an address or not:

```
If (Nil($pPointer))
   ALERT("Nil pointer")
Else
   ALERT("Not a Nil pointer")
End if
```

## V6 Resolving Pointers

4D V6 introduces the **RESOLVE POINTER** command. This command accepts any pointer and returns either the string value of the variable or array the pointer references, or the table and field number of a pointer that references a structure object.

With **RESOLVE POINTER** you can treat an object's name as data. Imagine that you have a series of objects on a form, each of which calls the same project method. A series of objects used on a calendar or calculator are common examples. Each object calls the same method, but passes a unique number to identify which object made the call. The variables in this example are v0…v9. Prior to 4D V6, you needed to write a unique method for each button:

```
  ` v1 method:
MANAGE_CLICK(1)

  ` v2 method:
MANAGE_CLICK (2)

` And so on...
```

This seems like a waste of time since the object name contains the necessary information. With 4D V6 and **RESOLVE POINTER**, you can pass an object pointer, resolve the pointer, and use the name or portions of the name as data. Each object has the same method:

```
MANAGE_CLICK (Self)
```

The *MANAGE_CLICK* routine takes care of figuring out which object made the call:

```
C_POINTER($1;$pObject)

C_STRING(31;$sObjectName)
C_LONGINT($lTable)
C_LONGINT($lField)
C_LONGINT($lObjectNum)

$pObject:=$1

RESOLVE POINTER($pObject;$sObjectName;$lTable;$lField)

$lObjectNum:=Num($sObjectName) ` I.e. Num("v6")=6
```

If you have a group of objects with the same action that need to identify themselves, this strategy saves a lot of time when constructing or updating form code.

## Conclusion

Once you understand how to use pointers and what their limitations are, it is easy to use them effectively. Pointers allow you to write code that is not object-specific, and can therefore be reused in different situations without requiring modification.