PerfectScript Array Theory
Beginning to Intermediate

by Mike Fitzhugh
mlfitzhu@wustl.edu

Contents

About This Document

PerfectScript Array Theory: Beginning to Intermediate is primarily aimed at amateur macro writers who've grown too big for their old britches but whose new ones still drag on the ground and trip them up—people who score some initial successes with macros (who have, for example, worked their way through Doug Loudenback's excellent *Common Person's Macro Manual*) and then get ambitious, only to be given a rude awakening by intransigent arrays. The context for this exploration of arrays is PerfectScript, the WordPerfect macro language. Syntax aside, much of this material applies to arrays in programming languages other than PerfectScript, but I don't have a computer-science degree and my knowledge of other languages is spotty; if you're interested in general array theory, this may not fulfill your needs.

In the Beginning

Even for experienced PerfectScript coders, arrays can be intimidating, and rightly so. They have these extra things attached to them—brackets—that look like prison bars. And inside the pokey are mysterious numbers that can change from place to place in a single script, like different jailed felons suddenly popping up and displacing one another in the same cell. It's all very cloak-and-dagger. If you've tried to use an array and failed (not an uncommon occurrence), you end up even more mystified; the temptation is to throw up your hands and turn your back on the whole shebang, especially if you're not a professional programmer whose livelihood depends on mastering such concepts. But arrays are worth understanding, even for an amateur. They can make certain tasks much easier to program than they'd otherwise be, and they can cut down your program's size by quite a bit in some circumstances.

However, I probably don't need to say any more on that topic. You probably already know that arrays will pay back the time you invest in them, or you wouldn't even have started reading a document about them which, at approximately 90 pages, is essentially a short book. So I won't dish out any more hype, especially since this document omits discussion of the practical, productive kinds of tasks you can accomplish with arrays. That's a tutorial's job, and this isn't a tutorial. Why not? Because arrays are so complex that in order to get much out of a practical tutorial, even a beginning-level one, you need to understand them theoretically first. If you don't, odds are that you won't be able to successfully get through the tutorial. You can make all sorts of mistakes with a programming technique if you don’t understand how it works and why it performs as it does. Tutorials aren't likely to give you all the information necessary for such understanding, especially concerning a technique as conceptually abstract as arrays. The intention behind this document, then, is to present a discussion that focuses not on *what* you can do with arrays but on *how* you should do it, as well as on how arrays themselves do things and how humans can understand them.

So let's dive in. When it comes to arrays, the PerfectScript manual (chapter 3) says, "Variables point to a memory cell where data is stored. . . . Adjacent memory cells, associated by a name, are collectively called an array. Use Declare, Local, Global, or Persist programming commands to declare an array name and the number of cells, or elements, associated with it." That's the kind of gobbledygook which only becomes clear after you learn about arrays somewhere else.

So what are they? The short answer is that an array is a *multiple variable*. Uh, right. So let's get started on the long answer. You're already used to normal variables with single values. For example, in a script, you might—since arrays are scary—type **`vMonster := "Dracula"`**, and from then on **`vMonster`** holds the string value **`Dracula`**. You can think of a regular variable like **`vMonster`** as a file folder holding a text document (which in this case is **`Dracula`**). But **`vMonster`** and any other regular variable can only hold a single value: one string, one numerical value, one boolean. You can't put more than one thing in it. An array, on the other hand, is a variable that can hold multiple values. It does everything a normal variable can, but also has an extra feature that enables it to transcend the limit of a single value that binds normal variables.

This is where those forbidding brackets, the "prison bars," come in. Trust me, they'll become your friends, opening doors rather than closing them. The brackets are what makes it possible for arrays to hold more than one value. When you declare an array, you decide in advance how many different values you want the array to be able to hold, then you put the corresponding number into the brackets when you do the declaration. For example, if you wanted an array called "arMonsters" to have the names of four monsters in it, you would type **`declare arMonsters[4]`**.

Here's where you can start getting mixed up. In most scripts, the next step would be to do what programmers call "populating the array," that is, to give it those four values. But you can't just dump those values willy-nilly into **`arMonsters[4]`**. What the scripting engine does when you declare an array with four values is to create four predefined slots, each slot set apart from the others by its digit (1, 2, 3, or 4). Each monster name needs to be precisely inserted into *one* of those slots. The slots are denoted by a combination of several features: a) the array name, "arMonsters" in this case; b) the brackets; and c) inside the brackets, the number of the slot. So the first slot is **`arMonsters[1]`**, the second slot is **`arMonsters[2]`**, the third is **`arMonsters[3]`**, and the fourth **`arMonsters[4]`**. Each individual slot works exactly like a normal variable, holding only a single value.

If you're still thinking of your array as a corporate body, one big pot of coding soup, it might seem logical to just populate your array all at once by typing something like...

```
arMonsters[4] := "Dracula" + "Wolfman" + "Blob" + "Killer Tomatoes"
```

...but you won't achieve your intended result. Because you've essentially created four normal variables that simply happen to be similarly named, **`arMonsters[4]`** will have the value "DraculaWolfmanBlobKiller Tomatoes," while the first three slots remain empty.

"Now hold on a minute, Mike!" you might say. "Didn't you just tell me that **`declare arMonsters[4]`** creates the entire four-slot array? So why doesn't it hold all four values? Why is **`arMonsters[4]`** all of a sudden just a single slot holding one value? Why *isn't* it one big pot of coding soup? What gives?"

Yes, you're right to be confused. Here's the deal: **`declare arMonsters[4]`** only happens once. It's a unique occurrence, existing only at the moment of the array's declaration. Thereafter the array in its entirety is *no longer* **`arMonsters[4]`**, but rather "arMonsters" with *empty* brackets: **`arMonsters[]`**. (Or, if using a technique called "slice" syntax, it's **`arMonsters[..]`**, but that's a more advanced topic which we'll examine later). After declaration, **`arMonsters[4]`** no longer represents all four slots, but instead just the very last slot, just as **`arMonsters[1]`**, **`arMonsters[2]`**, and **`arMonsters[3]`** label their respective slots. Therefore to populate the array properly, you would do something like this:

```
arMonsters[1] := "Dracula"
arMonsters[2] := "Wolfman"
arMonsters[3] := "Blob"
arMonsters[4] := "Killer Tomatoes"
```

From this point on, each numbered slot functions just like a normal variable. You can overwrite the value of one of them exactly as you would with a normal variable (for example, at a later point in the script, you could do `arMonsters[2] := "Mummy"`), and it won't affect the others. And you can use them just like a normal variable, doing something like this...

```
Type ("The slimiest monster is the " + arMonsters[3] + ".")
```

...which will type out in the WP document, "The slimiest monster is the Blob."

You don't even have to give values to all the array's slots. You already know that you can declare a normal variable and just leave it empty; well, you can do the same with array slots. So instead of populating the entire array, you could do only `arMonsters[2] := "Wolfman"` and your macro will be none the worse for wear (although if you ever wanted to use slots 1, 3, and 4, you could of course have to give them values at some point).

One of the small annoyances you have to deal with when working with arrays is keeping track of which value you've put into which slot. When I'm writing a script, the way I do it is usually by keeping a hardcopy list by my mousepad. That way, if my macro is very long (and they do seem to lengthen out past the three-page mark with disheartening regularity), I don't have to refresh my memory by scrolling all the way back to the spot where I populated the array, thus losing my current cursor position.

The End of the Beginning: Getting Ready to Move Ahead

That's the lowdown on basic, no-frills, baby arrays. They get a lot bigger and fancier, though, and before continuing on, you need to know two things:

- **First**, what I've been calling array "slots" are officially known as "elements," and will be referred to as such for the rest of this document. To someone learning about arrays for the first time, a "slot" is much more intuitive than an "element" (the latter word conjures up painful thoughts of high-school chemistry). But now that you've grasped the concept, it's time to graduate to the correct terminology. `arMonsters[2]` is no longer a slot, it's an *element*, as are `arMonsters[1]`, `arMonsters[3]`, and `arMonsters[4]` as well.

- **Second**, in discussions concerning arrays, you'll run across other programmers using the term *index* and its plural, *indexes* (or the more traditional *indices*, which is what I use here). This "index" bit can get murky, so I include an entire section on it below, but for now just remember that the index refers to the number of each array element (i.e. the number of each slot). Thus, `arMonsters[2]` has an index of 2, `arMonsters[3]` has an index of 3, and so on.

OK. We've gotten progressively more complex, and while this document is supposed to offer a discussion of theory rather than provide a tutorial, now we need to stop and get some hands-on experience before these concepts become so abstract that you start to lose your grip on them. So we'll take a break from the theory and run through a brief mini-tutorial—the only one in this document. Please note that, except regarding knowledge of arrays, you are already assumed to have intermediate- to advanced-level PerfectScripting capabilities, so in what follows I don't explain anything but the arrays. If there are parts of the code unrelated to arrays that you don't understand, you'll need to look elsewhere for relevant information.

Let's say you want to write a macro that uses a dialog with radio buttons, then pops up a message afterward confirming the user's choice and dictating a particular course of action depending on whether users re-confirm or change their minds. The macro itself is whimsical, but it performs a type of task that you might indeed find yourself needing in a business environment. Here's the code for the whole macro, unformatted so you can more conveniently print it out and study it at your leisure. You can also cut-and-paste this code into a WP macro window, save as a WCM file, and run it to see the script in action (at least in WP 8; it hasn't been tested in other versions):

```
Application (WordPerfect; "WordPerfect"; Default!; "EN")

// Declare and populate the array:
declare arMonsters[4]
arMonsters[1] := "Vampire"
arMonsters[2] := "Abominable Snowman"
arMonsters[3] := "Mummy"
arMonsters[4] := "Blob"

DialogDefine ("ChooseMonster"; 40; 40; 192; 110; Sizeable!; "Choose Monster")

DialogSetProperties ("ChooseMonster"; FontName:"MS Sans Serif"; FontSize:8p)

DialogAddText ("ChooseMonster"; "Static1"; 15; 6; 108; 13; ShadowBox!+Center!;
"Please Select Your MONSTER")

DialogAddRadioButton ("ChooseMonster"; "RadioBttn1"; 25; 25; 100; 10;
arMonsters[1]; monster1; RadioAuto!)
DialogAddRadioButton ("ChooseMonster"; "RadioBttn2"; 25; 46; 86; 10; arMonsters[2];
monster2; RadioAuto!)
DialogAddRadioButton ("ChooseMonster"; "RadioBttn3"; 25; 67; 55; 10; arMonsters[3];
monster3; RadioAuto!)
DialogAddRadioButton ("ChooseMonster"; "RadioBttn4"; 25; 86; 54; 10; arMonsters[4];
monster4; RadioAuto!)
DialogAddVLine ("ChooseMonster"; "VLine8"; 124; 25; 75)
DialogAddPushButton ("ChooseMonster"; "OKBttn"; 138; 40; 32; 16; OKBttn!; "OK")
DialogAddPushButton ("ChooseMonster"; "CancelBttn"; 135; 80; 38; 14; CancelBttn!;
"Cancel")
```

```
DialogShow ("ChooseMonster")
//Error-check to make sure the user pushed "OK":
IF (MacroDialogResult =2)
        EndDialog()
ENDIF

// User may have pushed OK but still didn't select a monster; set a variable that can be
used to error-check for this possibility:
Declare vDidSelect:="no"

// When a dialog with radio buttons is dismissed, all the radio buttons return a value--if it
was selected, a radio button returns a numeric 1; if not, a 0. It's up to the macro writer to
figure out which returned a 1 amongst all those 0s:
FORNEXT (i; 1; 4)
        monsternum:=NumStr(i)
        vTheNumber:= Indirect("monster"+monsternum)

        IF (vTheNumber=1)
                // User did choose a monster. Unset the error-checking var so it won't
                generate the error message later:
                vDidSelect:="yes"

                // Now run the procedure that pops up the appropriate message and inserts
                the  monster chosen by the user:
                Call ShowMonster(arMonsters[i])
        ENDIF
ENDFOR

IF (vDidSelect = "no")
        EndDialog()
ENDIF

PROCEDURE ShowMonster(pTheMonster)

        DialogDefine ("ConfirmMonster"; 80; 80; 192; 55; OK! | Cancel! | Sizeable!;
        "Your Monstrous Choice")

        DialogAddText ("ConfirmMonster"; "Static1"; 15; 6; 160; 25; ; "You chose the "
        + pTheMonster + ". Is that really your favorite monster? Press OK if yes,
        CANCEL if no.")

        DialogShow ("ConfirmMonster"; "WordPerfect")

        IF (MacroDialogResult = 1)
                MessageBox(; "Your monstrous choice has been confirmed"; "Oh, how
                sweet--you really do like the " + pTheMonster + ".")
```

```
        ELSE
            MessageBox(; "You are a fence-sitting waffler!"; "The macro will now
            close, but you should run it again and choose the monster you truly love.")
            Quit
        ENDIF
    ENDPROC

    PROCEDURE EndDialog()
        MessageBox(; "Monster Protest"; "You didn't choose a monster. They feel
        neglected. The Guild of Malificent Creatures has filed a victim complaint. Shame
        on you!"; IconWarning!)
        Quit
    ENDPROC
```

Now let's examine the array portions of the macro. First off, we declare an array with the choices we'll want users to pick from, then populate it...

```
declare arMonsters[4]

arMonsters[1] := "Vampire"
arMonsters[2] := "Abominable Snowman"
arMonsters[3] := "Mummy"
arMonsters[4] := "Blob"
```

...and now we have an array ready for use.

Next, we put the array to work in a dialog box with radio buttons. The radio buttons take **`arMonster[]`**'s elements as their **`ButtonText`** parameters (which I've italicized so they'll stand out):

```
DialogAddRadioButton ("ChooseMonster"; "RadioBttn1"; 25; 25; 100; 10;
arMonsters[1]; monster1; RadioAuto!)
DialogAddRadioButton ("ChooseMonster"; "RadioBttn2"; 25; 46; 86; 10;
arMonsters[2]; monster2; RadioAuto!)
DialogAddRadioButton ("ChooseMonster"; "RadioBttn3"; 25; 67; 55; 10;
arMonsters[3]; monster3; RadioAuto!)
DialogAddRadioButton ("ChooseMonster"; "RadioBttn4"; 25; 86; 54; 10;
arMonsters[4]; monster4; RadioAuto!)
```

After this, the dialog pops up and the user (hopefully) chooses a monster. Now we have to do something with the user's choice. When a dialog with radio buttons is dismissed, all the radio buttons return a value. If one of them was selected, that radio button returns a numeric 1; if not, a 0. It's up to the macro writer to figure out which (if any) returned a 1 amongst all those 0s, and in this macro we'll do it in a **`FORNEXT`** with an **`Indirect`** (a technique is based on a demonstration in one of Corel's sample scripts). When we've worked our way through the dialog-variable concatenation and error-checking and we finally know (via the **`IF`**) that we have a 1

instead of a 0, we call procedure **ShowMonster()**, which directs the rest of the macro—a procedure that, not coincidentally, has our array as its parameter:

```
FORNEXT (i; 1; 4)
        monsternum := NumStr(i)
        vTheNumber := Indirect("monster" + monsternum)

        IF (vTheNumber = 1)
                // User did choose a monster. Unset the error-
                checking var so it won't generate the error
                message later:
                vDidSelect:="yes"
                Call ShowMonster(arMonsters[i])
        ENDIF
ENDFOR
```

How is this procedure call working? In the call, **arMonsters[]** doesn't have the kind of hard-coded element number that we've seen thus far; instead, the brackets contain an "i." But that's not a string! Look back up at the beginning of the **FORNEXT** and you'll be reminded that it's the loop's increment value. In other words, the array's index, its element number, is determined by the **FORNEXT**'s increment value. As the loop goes around, therefore, the parameter **arMonsters[i]** changes its value: the first time it's actually **arMonsters[1]**, which contains "Vampire"; the second time it's actually **arMonsters[2]**, containing "Abominable Snowman"; the third, **arMonsters[3]**, with "Mummy"; and the fourth and final time it's **arMonsters[4]**, "Blob"—so that each time, **arMonsters[i]** contains the text for the monster which corresponds to the number of the radio button that the loop is currently checking. Thus, when the **IF** fires because the radio button in question was selected by the user, the appropriate array element is sitting in the parameter slot, all ready to go, and the correct string value is thereby passed into the procedure.

The last part of the macro is the procedure **ShowMonster()**, which pops up various messages depending on what the user does. It's an important part of the macro, but as far as arrays are concerned, there's not really much going on. The procedure requires a string for its parameter, and that string happens to be passed into it via an array element, but once the procedure actually runs, all that matters is that the value did get passed in. The array plays no further part in the script.

But that's not to say that **arMonsters[]** hasn't already done a *monstrously* good job. I think it's time for some hype. (I know, I said I wouldn't do this, but I can't resist.) What if we hadn't used arrays? What if we had simply put strings directly into the radio-button **ButtonText** parameters (**"Vampire"**, **"Abominable Snowman"**, and so forth) instead of going through all the rigomarole of declaring the array and then using array elements in the dialog? We certainly could have done that. At first glance, then, it seems like using the array was actually counterproductive, requiring extra code at the top of the macro and not doing anything for us in return that we couldn't have done another way just as well.

But stop a moment to consider the code for the procedure call. If we didn't have **`arMonsters[i]`** to use as the parameter, how would we have passed the correct text string into the procedure? Here's one way. The following code could function as a direct replacement for the currently coded procedure call, with no modifications to any other part of the macro. However, it is obviously much more clumsy and undesirable than the elegant, short single line of code that the array makes possible:

```
SWITCH("monster" + monsternum)
        CaseOf  "monster1":
                Call ShowMonster("Vampire")
        CaseOf  "monster2":
                Call ShowMonster("Abominable Snowman")
        CaseOf  "monster3":
                Call ShowMonster("Mummy")
        CaseOf  "monster4":
                Call ShowMonster("Blob")
ENDSWITCH
```

In sum, there are certainly other ways to get the job done, but none that I can come up with are as lean and mean as the array. Arrays rule, arrays are cool, arrays are a snarfobulous coding tool!

Direct-Assignment Declaration

You now know how to declare arrays using the same method as that for normal variables, but there's a more efficient way-to-array in which you don't have to use the regular **`declare`** syntax: the direct-assignment method (I think of it as the "lump-sum" technique). This method requires the use of what is called an "array literal" or "array constant," which consists of a group of values—that's right, not elements, but instead the actual values—framed by braces (e.g. **`{"Dracula";"Abominable Snowman";"Mummy";"Blob"}`**). The nomenclature may seem strange, but it's really not too big of a jump from terms you already know: an array literal (constant) is just a bunch of normal literals/constants inside braces. (Remember, a string value like **`"abc"`** is called a "string literal" or "string constant," while **`10`** is a "numeric literal" or "numeric constant.")

When using the direct-assignment technique, you declare and populate your array all one fell swoop, using only a single expression; this is apparently why you needn't use **`declare`**. Instead, on the left side of the equals sign you just use the name you wish to give the array, and on the right, the braces with their values:

```
arMonsters := {"Dracula"; "Abominable Snowman"; "Mummy"; "Blob"}
```

That's all you need to know in order to begin constructing arrays by direct assignment, but for those who are interested, here's how the scripting engine processes this code: first, it hits **`{"Dracula";"Abominable Snowman";"Mummy";"Blob"}`**. It knows that the braces have to contain an array literal/constant—there's nothing else in the PerfectScript language that a

lone set of braces can do—so it creates a temporary array (let's call it a "temparray") in memory. The next thing it sees is **`arMonsters[]:=`**, whereupon it makes sure that it doesn't already have an "arArray" array hanging around. If it does, it merely passes the array constant into the array, overwriting whatever array structure and values were there, but if the array doesn't already exist, it *creates* **`arMonsters[]`** and then passes the temparray's data into it. And now that we have an array containing not just the data we specified but also the name we designated, the macro automatically discards the temporary array. (For further examples of array literals, see below, **Dimensionality Demasked | Conceptualizing Arrays | Using Arrays**, and the relevant subsection under "Building Arrays: Best Practice.")

Crossing into the Next Dimension

It's time to introduce another new concept. Once you've begun exploring this one, you're no longer an array beginner—you'll have moved into the staging area for intermediate-level work. As you might expect, it's the toughest yet: *dimensions*.

The baby arrays we've been talking about thus far are what's called "one-dimensional." Yes, you can have arrays that programmers describe as "two-dimensional," "three-dimensional," and even beyond—PerfectScript allows you to have arrays with up to ten dimensions! And, yes again, this is indeed as complicated as it sounds. The next twenty pages or so are dedicated to the details of array dimensionality. For now, what's important is that the two most common types of multidimensional arrays, 2D and 3D arrays, are generally described in geometrical terms: 2D arrays are talked about as tables, having rows and columns, while 3D arrays are thought of as cubes, with planes, rows, and columns. In all multidimensional arrays, the dimensions are spoken of as ascending from left to right, in the direction we read, so that the leftmost dimension is the first, the one next to it is the second, and so forth until the rightmost dimension, which is the last.

That's enough for now on multi-dimensional nomenclature; we'll revisit the topic later. Let's get more practical and talk about how to declare a multi-dimensional array. If you want an array to be multidimensional, you have to declare it that way right at the outset, and to do that you must put more than one number inside the brackets: one number for each dimension, with each number separated from the others by a comma or semi-colon. (I prefer semi-colons.) So if you wanted **`arMonsters[]`** to be two-dimensional, you'd declare it like so:

```
declare arMonsters[3;4]
```

Now **`arMonsters[]`** has three elements in its first dimension and four elements in its last dimension. The elements in the last, rightmost dimension will always hold the array's values; the elements in the other dimensions specify multiple sets of the last dimension's elements. Lost? Don't worry, you should be. Keeping track of multidimensional array organization—let alone successfully using such an array—is an adventure all by itself, rather like Perseus in the minotaur's labyrinth. Later on, I'll give you a spool of cyberthread and boot you inside. For now, we'll merely open the door to the maze and content ourselves with a glance inside, confining our discussion to two-dimensional arrays.

In a 2D array, how many sets of values the first dimension gives to the second is denoted in the first dimension's index number. We declared a "3" for the first (left-most) dimension of our new **`arMonsters[]`**. That **`3`** gives the array three sets of however many elements were specified in the second dimension. That, of course, was 4, so **`arMonsters[]`** has three sets of 4 second-dimension elements. Thus the second dimension has 12 elements in all—in other words, it can hold 12 different values.

To get to these values in the last dimension—or, rather, to get to these empty elements in the last dimension (you haven't put any values into them yet)—you have to go *through* the first dimension, just like you have to go through the folder hierarchy on your hard drive to get at the files inside. If you have two nested folders, you have to go through both of them to get to a file; likewise, if you want to put a file inside, you have to open up both folders in succession.

To illustrate this as it applies to our 2D array, let's put a value into one element of each of the three sets. (We'll leave the other elements empty for the sake of convenience.) First, let's assign "Dracula" to the first element in the first set...

```
arMonsters[1;1] := "Dracula"
```

..."Wartpuckered Hobgoblin" to the fourth element in the second set...

```
arMonsters[2;4] := "Wartpuckered Hobgoblin"
```

...and "Monster under the bed" to the second element in the third set:

```
arMonsters[3;2] := "Monster under the bed"
```

You can see how in each case we open up our desired set of values by designating the correct "folder" from the first dimension in the array (if we want the first set, we type "1" in the first dimension's place; if we want the second set, we type "2" in the first dimension's place; if we want the third set, we type "3" in the first dimension's place). Only then do we get to the array elements that can actually hold values (1, 2, 3, or 4 from the second dimension).

Remember, too, that everything else is still empty. So while this...

```
Type("The " + arMonsters[2;4] + " is a little-known sprite that specializes in pinching an embarrassing portion of human anatomy")
```

...will successfully type out "The Wartpuckered Hobgoblin is a little-known sprite that specializes in pinching an embarrassing portion of human anatomy", this...

```
Type("The " + arMonsters[2;3] + " is some other kind of monster")
```

...will result in a macro error because we didn't give the second set's third element a value yet; the only element in the second set that has a value is the fourth one.

At this point I recommend letting your grasp of arrays percolate and settle further by visiting Julie Jeppson's WP macro site, which offers the only PerfectScript array tutorial available on the web (as of this writing) at http://home.comcast.net/~jj84097/wpmacros/psjournal/apr98feat.htm. Unfortunately, it's nearly as brief as the mini-tut on one-dimensional arrays that you've worked through here, but even if Julie's explanations are rather terse, at least her offering serves up a multidimensional array for analysis. With the groundwork we've laid, you should be ready to handle that. As you study Julie's code you'll probably need to refer back to our discussion, but that's all to the good—the process will help cement your knowledge and make you better prepared to move to full-fledged intermediate-level efforts. (Warning: before heading over to Julie's tutorial, brush up on your callbacks.)

And don't be worried if much of the material is still rather hazy. We're about to re-examine the more confusing topics in greater detail than before. Strap in!

Some Windex on Index

We've briefly mentioned that arrays have *indices*; these are used to access its *elements*. Unfortunately, because each element is accompanied by an index, it's easy to confuse the element with its index. But they aren't the same thing.

While we'll examine elements further in the next section, in order to understand indices we need to recognize that an "element" is kind of like one of those big, accordion-style file folders; it's a data container into which you can insert other data containers (like regular file folders) or actual records (single pieces of paper). An *index*, on the other hand, is like the label we attach to a file drawer or folder; in an array, the index is a bunch of numbers attached to the data containers in a dimension. You use those numbers to access the data, just like you read the labels on a file drawer or folder to find the material you're after. In other words, you might think of an index as an "identifying number," just as a file folder's label is an "identifying word." And if indices in an array are only labels, what are they labeling? The actual "file folders" and "files" themselves—the array's elements.

Another way to think of this is to liken an element to a Windows folder—just as a folder on your hard drive holds either another folder or a real file, an array *element* holds either another structural unit or a real value, while the element's *index* is its folder name or filename.

Yo, Dude! Say it Right!

Unfortunately, clearing up the confusion surrounding "element" and "index" doesn't get us completely in the clear. Another reason the word "index" is so messy is because it's used in two different ways:

- To denote the identifying number of each individual element *within* a dimension (i.e. "the index of the fifth element in the first dimension is 5"). This is the usual meaning of

"index," the one we just got through discussing above. Array elements will always be tagged by a number, commonly called the element's "index," and that index will always be determined by the element's position in its dimension. Is the element first in line? Then its index is 1. Is the element fifteenth down the list? Then its index is 15. We can't name the first element "FirstElement" or the fifteenth "CoolestThingInMyArray"—each will always and ever only be accessible through the index numbers 1 and 15, respectively. (Note that in other programming languages there exists a structure called an "associative array" or "hash" in which such descriptive naming is possible, but PerfectScript has not implemented this feature).

- To denote the *entire group* of identifying numbers in a dimension, *as a whole* (i.e. "there are five columns, so the column index is 5" or "I specified six rows, so the row index is 6"). However, "index" really shouldn't be used this way; technically, if you're talking about a whole group of indices, it stands to reason that you should use a term that denotes plurality. And one does in fact exist for this usage: to denote an entire array dimension, you're actually supposed to speak of that dimension's *index* ***range*** rather than just its index: "I specified six rows in this array, so the *range* of the row index is 6." But experienced programmers don't need correct terminology to sort out such things; they can tell the difference by context, so they tend to get sloppy with their wording because they know that other experienced programmers won't be misled. We newbies, on the other hand, will definitely be confused, but being aware of this problem and watching out for it can cut down the head-scratching.

Indexing the Elements

Indices are good for more than simply getting to array elements, by the way. You can also use them to manually figure out how many elements an array has: simply multiply all the index ranges together. For example, **`arArray[5;2;4]`** has 5 planes, 2 rows and 4 columns, for a total of 40 elements. Structurally, the real concrete data—the values—of an array reside in the last dimension (**`4`** in the current example), so the most accurate way to think of this is "(5 x 2) 4 = 40". In other words, the first dimensions determine how many 4s there are. In **`arArray[4]`**, with one dimension only, there are only four pieces of actual data. But in the 2D array **`arArray[2;4]`**, there are eight pieces of concrete data (two sets of **`4`**). Likewise, **`arArray[5;2;4]`** has 40 pieces of data (ten sets of **`4`**). But more on this in the next section.

If this made little sense, don't despair. Soon, in the section, "Dimensionality Demasked," digital seer and compu-psychic extraordinaire JDan the Great reveals all!

The Hell-ament of Element

If you're confining your array-programming efforts to one-dimensional arrays, the concept of "element" won't cause you much trouble: a unidimensional array element always holds a value, just like a normal variable.

That's not the case with multidimensional arrays, however. Ulp. Yes, it's time for some growing up. We have to move beyond the quick-and-dirty beginner's rule of thumb that always equates elements with values. Digging a bit deeper reveals an element for what is truly is: a given unit of *space* (the space that's being tagged by the index) in any *single* array dimension. An "element" is actually less like a normal variable than it is like a folder—just as a folder on your hard drive holds either another folder or a real file, an array element holds either another structural unit or a real value. So if you've got a five-dimensional array, each of the five dimensions has "elements": for example, the first dimension of **`arArray[6;3;8;5;2]`** has 6 elements. Each of the first dimension's six elements acts like a folder, holding three of the units from the second dimension. (For more on this hard-drive-folder way of thinking about arrays, see the next section, "Dimensionality Demasked," subsection "Conceptualizing Arrays | Using Arrays".)

That last sentence might have raised a red flag for some readers. If the first dimension has 6 elements, then shouldn't the second one have 3, the third one 8, the second one 5, and the first one 2? That's what the array itself *says*, fer cryin' out loud, with the index ranges 6, 3, 8, 5, and 2: each dimension has a number of elements equal to its index range—right? Yet if what I said is correct (that all six elements of the first dimension *each* hold three units from the second dimension) then what the array's numbers—those nice, friendly digits 6, 3, 8, 5, and 2—seem to be saying is, in fact, *not* true, because the second dimension has to have a total of 18 elements (three elements in every single one of the first dimension's elements).

That's right. That's the way it is. Only the first, topmost dimension in an array—the one furthest to the left—is numerically transparent, having a number of elements equal to the number given by its index range. What actually determines how many elements are in a given dimension isn't just the dimension's index range, but also the *previous* dimension's total elements. The current dimension's index range does not give the total number of elements in the dimension, but rather the total number of elements that go inside each element of the previous dimension. Thus, to get the second dimension's total-element amount, we have to multiply its index range by the first dimension's index range: 6 x 3 = 18.

And that, of course, is not all. We can't restrict our multiplication to the previous dimension; i.e. we can't go down to the third dimension, do 3 x 8 = 24 and decide that the third dimension has 24 elements. Well, we could, but we'd be wrong. Remember, the second dimension doesn't just have three elements—it has 18! So instead, we have to do 18 x 8, for 144 elements in the third dimension. And how do we know that the second dimension has 18? Because we multiplied *its* index range by the index range of the first dimension, which is where the buck stops in this particular array. That's the key: the buck stops here. To find out how many elements the other dimensions have, you must multiply them by the index ranges of *all* the preceding dimensions, going all the way out to the first dimension to where the buck stops, or as Wittgenstein might put

it, to where this particular array "turns its spade" and stops digging. So to find the total elements of the third dimension, we do 6 x 3 x 8; for the second dimension, 6 x 3 x 8 x 5; and finally, to get the total amount of elements in the fifth, last dimension—the entire array—we do 6 x 3 x 8 x 5 x 2.

Let's say it again: each of a dimension's elements holds a single set of elements from the *subsequent* dimension, but that set is not the total number of elements in that subsequent dimension—rather, the set comprises the number of values denoted by that subsequent dimension's index range. The total number of elements in that subsequent dimension depends on the number of elements from the *preceding* dimension. And the total number of elements in that dimension depends on how many there were in the dimension that preceded *it*, and so on until the first dimension is reached.

Unless you're one of those people who always got an A in math, you can see why I called this section "The Hell-ament of Element." Perhaps approaching the issue in a somewhat different manner might shed more light on it. We'll take `arArray[6;3;8;5;2]` from the top again; however, this time we won't look back to the preceding dimensions. Rather, we'll look ahead to subsequent dimensions, doing the multiplication early, before we move on to each subsequent dimension. Ready? The first dimension has six elements; that's it, no complications, no perplexity, no multiplication necessary. But multiplication will be necessary for the second dimension, so while we're still here in the first dimension, let's take care of it. There are six elements in this first dimension, and each one of them contains three elements from the second—three because the second dimension's index is 3. So the second dimension totals 18 elements. And the same "container" principle applies to the rest of the dimensions as well. Each of those 18 elements from the second dimension contains a set of 8 elements from the third dimension (because the third dimension's index is 8), so the third dimension has a total of 144 elements. In turn, those elements each contain a set of 5 elements from the fourth dimension (since 5 is its index), and the fourth dimension ends up with 720 elements. Finally, The 720 elements from the fourth dimension each hold a set of two elements from the fifth (and last) dimension, so the last dimension has a total of 1440 elements.

Here's a helpful way to visualize this in action. When constructing an array, imagine PerfectScript using a dimension's index range number to construct a bunch of "element sets," then stuffing one of those sets into each element of the preceding dimension. So when building the second dimension—the one with `3` as its index range—of our current example `arArray[6;3;8;4;2]`, PerfectScript...

> **1.** ...first looks at the index range of the preceding dimension, `6`. Since the first dimension is, uh, first, PerfectScript knows that the index range transparently displays the number of elements in the dimension, so it needs do no further calculation: the buck stops there.

2. Having seen that it has six empty slots to fill with sets of elements from the next dimension, it turns to that next dimension (the second). Once it gets there, let's imagine that PerfectScript sees a basket holding a bunch of elements. It immediately looks at the accompanying index range, **3**. "OK," it thinks to itself. "Each set of elements from this dimension has to have three elements in it. Great! Lemme grab three elements, and...." The macro engine then grabs three single array slots (elements) from that second-dimension basket, then reaches back to the first dimension, opens up its first element, stuffs in the three elements from the second dimension, then closes up the first dimension's element.

3. "Arrighty then!" PerfectScript congratulates itself. "But oops, how many sets was I supposed to make again?" Then PerfectScript remembers that the first dimension had 6 elements in it. "Oh yeah, six sets!" it exclaims. "One down, five to go!" Opening up the first dimension's second element with one hand, it scoops up three more elements from the second-dimension basket with its other hand. After inserting them into the first dimension's second element, PerfectScript closes up that element.

4. PerfectScript then repeats this process all over again until six sets of three second-dimension elements have been created, and each one of the first dimension's six elements has been filled with one of the sets from the second dimension (one package of elements from the second dimension for each element in the first). Wa-la—the second dimension of the array is fully constructed.

5. "Now," PerfectScript mutters to itself, "on to the third dimension..."

Dimensionality Demasked

When PerfectScript documentation and programming gurus on webforums talk about array "dimensions," it's likely to make a normal programming newbie think of *Twilight Zone* reruns or reading *Wind in the Door* as a kid rather than of writing a WordPerfect macro. Honest, the word isn't Corel's fault. Long, long ago in the misty dawn of cybertime when great dinosaurs such as UNIVAC walked the earth and the First Arrayer first declared an array, the Primal Programmers decreed that this new, multivalent kind of variable be described in programming documentation as if it were a geometrical construct extended into space. They started with the least complex kind of array and described it as having only a single dimension, while the kind of array only slightly more complicated was dubbed "two-dimensional," and so on.

Macroduck in Mathematicsland

Programmers usually refer to a one-dimensional array as simply a "list" or "vector." Following the geometrical metaphor, if they have an array with two index ranges (two dimensions), they speak of it as a table, calling the first range (the leftmost range) the "row" designator and the second range (the rightmost) the "column" designator, or vice versa; in some programming

languages, it's the other way 'round. But no matter what programming language you're using, if there are three dimensions, it’s common to speak of the first one as a "plane" alongside the row and column. (After that—four or more dimensions—programmers don’t have any common terminology, but since 4D+ arrays aren’t used all that often, common terms for them aren’t in a whole lotta demand.)

At first glance all this seems harmless enough, but stop a moment to ponder the difference between "one-dimensional" and "two-dimensional" arrays. **`arArray[3;4]`** has three "rows" and four "columns." So far so good. And if the last index range in any array refers to columns, then a unidimensional array comprises nothing but columns, right? Accordingly, if we take away the "row" from our array (ending up with **`arArray[4]`**), we now have four columns—and that’s it. Each column holds a piece of data.

But hold on. There’s a conceptual problem here. If you have something that you’re calling a "column," then (in a mathematical context) it *must* be accompanied by a row. In math, you can’t have rows without columns, nor columns without rows. Even if a table has only a single "container"—in WP, just one rectangle sitting there on the page—it still has rows and columns (one of each) because it’s a table, not a real rectangle such as a Text Box. So if arrays are like tables, a one-dimensional array can’t actually be one-dimensional! If it has columns, it must also have rows, so **`arArray[4]`** doesn’t simply have four columns and nothing else. It must also have rows, one to be exact. It could therefore also be written as **`arArray[1;4]`**, but for the sake of convenience usually isn’t.

Hmmm, you say. Maybe this is one of those optional-parameter scenarios where the Corel documentation says that a particular command has a parameter that’s "optional," only it isn’t truly optional in the sense of being unnecessary for the command to run; the parameter *is* actually necessary, but you don’t have to specify a value for it because if you leave it out PerfectScript will automatically insert a default value. Here, then, perhaps PerfectScript inserts a **`1`** if you don’t specify a row index range.

Bzzzt! Wrong answer. If you declare **`arArray[4]`**, you must thereafter refer to the elements in the array according to their direct index, **`[1]`**, **`[2]`**, **`[3]`**, or **`[4]`**, but if you declare **`arArray[1;4]`** you can’t do that; you have to refer to array elements by specifying both rows and columns (**`[1,1]`**, **`[1,2]`**, **`[1,3]`**, and **`[1,4]`**). So even though both of these arrays would have exactly the same pieces of data—i.e. the same array elements—no more and no less, and both in the columns index range, PerfectScript itself certainly isn’t treating this syntax as one of those "optional-parameter" situations. No "optional" default row is inserted if you fail to specify it yourself. The array either has a row, or it doesn’t. What the heck?

Multi-Dimension Dissension

The problem doesn’t lie with PerfectScript, but stems from the way arrays are described by programmers. Although the phrase "one-dimensional array" might conjure up thoughts of a flat novel with shallow characters, these lowliest of arrays bely the comparison, but not because

they’re actually two-dimensional. They really are unidimensional. And because they brashly proclaim that fact, they prove themselves the most honest of arrays, for all other arrays are one-dimensional too—even those that supposedly have ten dimensions!

What?

It may come as a shock that the GPGs (Great Programming Gurus, the high priests of the ancient civilization of the primal programmers) have misled us these many years, but at least they thought they were doing it for our own good. Anyway, it can help a great deal to get past Big Brother’s math metaphors and take a peek at what’s really going on under the hood. In this case, the hood is opened by J. Dan Broadhead, a longtime developer of PerfectScript who hangs out at http://www.wpuniverse.com, affectionately known there as "JDan" (his username). "Internally, *all* arrays are one-dimensional," he says. "All the elements, regardless of the number of dimensions, are located sequentially one after another as a single long ‘list’ or ‘vector.’" Indices are just a convenience, a formalized way of tagging the elements that allows humans to mentally organize arrays in complicated patterns to perform complex programming functions. When a running macro reaches an instruction specifying a given array element, PerfectScript looks at the indices you typed, then does a quick little tap-dance, says JDan, "to figure out where the specified element is really located in memory relative to the beginning of the array." (This is called an "index-to-location" computation).

JDan helpfully explains the technicalities step by step. "Each index of any array has a range," he begins. "The range always begins at 1 as the lowest value, and extends to some upper value that you specify when you declare the array. The elements in the array are laid out in straight linear sequence. The rightmost index changes the fastest; each leftward index changes ever more slowly." Let’s say that you declare array **`arArray[3;4]`**. The right index has a range of 1 to 4—it contains four values. The left index range is three, so the first dimension ranges from 1 to 3, meaning that it has three elements. Each of these three will contain four values, as specified by the second dimension, so there’ll be 12 actual values in all.

After reading the expression "declare arArray[3;4]", PerfectScript goes to your computer’s RAM to construct the array, laying the array elements out in the following order:

arArray[1;1]
arArray[1;2]
arArray[1;3]
arArray[1;4]

arArray[2;1]
arArray[2;2]
arArray[2;3]
arArray[2;4]

arArray[3;1]
arArray[3;2]
arArray[3;3]
arArray[3;4]

See how the index on the right changes with each element, but the left index doesn't? The rightmost index in any array will always change as rapidly as its range permits. Only after the right index has ranged through all its values does the left index change, and it always changes by simply adding one. Concurrent with this change, the rightmost index starts itself all over again.

If a programming language changes the rightmost index first when laying out an array in system memory, the array organization is called "column-major order." Other programming languages do it differently, storing the elements in memory with the leftmost index changing first, moving to the right instead of to the left. This is called "row-major order," since the row number changes first rather than the column. However, this has to do only with how arrays are laid out internally in RAM. None of it really matters for programmers coding in these languages; it doesn't affect array syntax or how arrays are used in any way. The rightmost dimension is still called

Even if you have a super-duper Dr.-Frankenstein-mad-scientist complicated ten-dimensional array, the same simple principle obtains, simply recurring for each index from right to left. "In the macro debugger, expand the elements of an array variable to see its contents," JDan suggests. "You'll see the elements in this order, with the rightmost index changing first."

In the computer's RAM, however, the array elements are not recorded in array syntax; that is, as PerfectScript walks down memory lane, it *doesn't* see the array elements laid out as presented by the macro debugger. What PerfectScript sees, instead, is this:

arArray1
arArray2
arArray3
arArray4
arArray5
arArray6
arArray7
arArray8
arArray9
arArray10
arArray11
arArray12

So how does the macro system actually keep track of array elements? Let's say that we want to use element **`arArray[2;1]`** (row 2, column 1 if you're thinking in geometrical terms) in an imaginary macro. That value is actually the fifth item in memory, located directly after all the

elements of the first group (row 1), which comprises the first four items. PerfectScript reads the (imaginary) macro we wrote and uses the indices **`[2;1]`** to get to the fifth item in the array by multiplying the left single index (2) by the entire range of the right index (4). First, however, it subtracts 1 from that left single index, so before it does anything else here, it performs the arithmetic operation 2 - 1. (For mathematical reasons beyond the scope of this discussion, this is necessary to avoid overshooting the mark.) Finally, after the subtraction and subsequent multiplication, PerfectScript, adds the left single index to the mix. Although the computer does this in binary form, the three-stage computation looks like this in decimal notation:

```
( (2 - 1)  *  4 ) + 1
```

The answer is 5. Exactly what we need, although we don't know that—and we don't need to know it. PerfectScript computes 5, then obediently extracts the fifth element and puts it into the macro as we asked it to, while we sit there thinking about "row two, column one."

JDan reviews the process. "The computation is to multiply the row number (minus 1) by the full range of the column numbers, and then add the column number," he says. The result is a formula that PerfectScript uses to locate array elements. If **`rowItem`** is the row index, **`columnItem`** is the column index, and **`columnRange`** is the range of the column index, we end up with:

```
( (rowItem - 1)  *  columnRange ) + columnItem
```

When confronted with arrays of more than two dimensions, the macro engine merely extends the formula leftward, multiplying each further index to the left by the range of the index to its right. So in a 3D array, with **`planeItem`** as the index of the plane and **`rowRange`** as the range of the dimension that the row index is in, the formula would be:

```
(((((planeItem - 1) * rowRange) + rowItem) - 1) * columnRange) + columnItem
```

Thus, for an array declared as **`arArray[2;3;4]`**, the element **`arArray[2;2;3]`** is located at **`((((2 - 1) * 3) + 2) - 1) * 4) + 3`**, or location 19. "Looks messy doesn't it?" JDan observes. "That's why the macro system does it for you."

Conceptualizing Arrays

It's nice that you needn't worry about getting your hands dirty with all this, but it's helpful to understand how it works; if you know that "dimensions" aren't really dimensions when you get right down to them, you can avoid the confusion that sometimes afflicts neophytes who use the "plane, row, column" terminology to define for themselves exactly how arrays work. And it can get pretty confusing.

Why Bother with all that Theory?

By giving a detailed example of a conceptual mistake I made which led to a bug in one of my macros, this subsection demonstrates how a wrong theory of arrays can have serious consequences. (My woes may also bore some folks to death; if you trust me that the theory is important, then by all means skip this section and go on to the next.) If you think that the macro system itself builds arrays as geometrical constructs, you'll be prone to think about arrays in a rigid, lockstep fashion, insisting to yourself that the rightmost index always has to be a column. Like me, you'd have to assume that, as I wrote in an earlier draft of this document (before JDan corrected me), "the two rightmost indices of all arrays work according to a table model. No matter how many dimensions an array has, the last two dimensions are, respectively, rows and columns."

Well, that's nominally true. Programmers will always speak of the two rightmost dimensions as rows and columns. But that's only for the convenience of people; the programming language itself does not think of arrays that way. However, I thought that it did. Adherence to this rigid "table model" led me to further conclude that "the term 'one-dimensional' is actually false: there isn't any such thing as a 'one-dimensional array.' All arrays are actually multi-dimensional. So '1D' arrays really have *two* dimensions. They, too, are tables." After all, by definition all tables have both columns and rows. If it doesn't have both, it's not a table, and that's important to recognize because it has practical ramifications. For example, consider the difference between WordPerfect boxes and tables. You can insert a text box and type something inside, or you can open the table dialog and make a single-column, single-row table that looks like exactly the same, typing that same text inside. You can even give the two "boxes" the same style of outline. But they are not at all the same thing under the hood. The table is no simple box; it is a full *table*, defined by one row and one column. This ontological status determines the table's "epistemology"—how it works. Size is just one example: you resize a text box with the Box Size dialog, but you can't do that with a table. For size purposes, WP treats tables as a special, self-contained group of pages-on-the-page (as opposed to a box on a page); instead of right-clicking and going to the Box Size dialog, to change table size you have to right-click on the table and get **Table Tools | Format | Table**. Despite the fact that they can look the same, a text box and a single-row, single-column table are not at all the same thing.

OK, I thought. So tables are their own animal and always have both rows and columns. That's important. Fine. So far, so good: if arrays are geometrical constructs, they are always at least two-dimensional. But how to explain the fact that you can declare an array without two index-range numbers, e.g. **`arArray[3]`**? Shouldn't that array actually be **`arArray[1;3]`**? Yes, I decided—under the hood, the first example must really be the second one, because otherwise **`arArray[1;3]`** would be a table, but **`arArray[3]`** wouldn't: it would be merely three separate boxes, not bonded together by anything. But if all single-dimensional arrays were tables, that was impossible, so it couldn't be true. **`arArray[3]`**(with 3 "columns") and **`arArray[1;3]`**(one formally declared "row," three "columns") had to be exactly the same thing. And since that was the case, the index-range syntax for a so-called "one-dimensional" array (**`[3]`**) had to be "optional" in the sense that many parameters of PerfectScript commands are "optional"—they're absolutely necessary for the command to work, it's just that you don't have

to specify them when you write a macro. If you leave them off, PerfectScript itself inserts a default value into the parameter that was left blank. Therefore, I thought to myself, whenever someone declares an array with only a single index range (a "column"), PerfectScript must be stepping up and filling in the blank (the "row") with a default value of **`1`**.

This is where the rotten fruits of my misguided labors started to get noticeably overripe. If **`arArray[3]`** and **`arArray[1;3]`** are in fact the same thing because PerfectScript automatically converts the former into the latter, then it stands to reason that no matter which way you declare the array, you can refer to its elements with *either* the single-index or double-index syntax. Thus, if you wrote **`declare arArray[3]`** and then wanted to retrieve the value of the second array element, you could type either "arArray[2]" or "arArray[1,2]." This is incorrect. It's true that the two arrays are essentially the same—they could both have the same three values in their "columns"—but not for the reasons I thought. As we have already seen in the section "Multi-Dimension Dissension" (above), PerfectScript arrays don't actually follow a geometric schema. The number of indices you have to use is a direct function not of geometry but rather of simply how many indices you yourself declare in the first place. And now that we understand that, we're ready to learn the reason why PerfectScript treats **`arArray[2]`** differently than **`arArray[1,2]`**—even though they really are exactly the same thing: it's just the way PerfectScript was structured by JDan and his team. "Once you declare an array with a certain number of indexes (dimensions), with certain ranges," JDan warns, "then you must always use that same amount of indexes to refer to an element of that array." In other words, after an array is declared, you must always refer to its elements via the syntax used in the array's declaration. If you gave it both a "row" and a "column," you must always use a "row-column" index (i.e. use two numbers); likewise, if you give it only a "column," you must always use just a "column" index (with only a single number). **`arArray[3]`** will always require you to refer to its elements with a single index, while **`arArray[1;3]`** will require you to use two indices. To access the second array element of these arrays you would specify indices **`[2]`** and **`[1,2]`** *respectively* rather than interchangeably.

In sum, if you didn't know how arrays work under the hood, your conceptualization of arrays might be much more contorted than necessary and could lead to writing syntax bugs in your macros. My episode of error offers a perfect example of why theory is sometimes downright necessary for users of technical instructions—and of how inadequate Corel's macro documentation is. Great program, powerful macro language, and cruddy help for customers who don't want to exchange an arm and both legs for a special support package.

Using Arrays

Well, sure, it's great to understand how PerfectScript really constructs and manipulates arrays under the hood, but as JDan notes, it's complicated and messy. To use arrays we have to conceptualize them, and our conceptualization needs to be more easily thought about than what really goes on in RAM; otherwise, none of us would ever use them. So how should we think about them?

"You can say that the array **`arArray[3]`** has 3 columns if you want, thus continuing the notion that the rightmost index is always the columns," JDan says. "But with one-dimensional arrays, I drop the row/column concept, and just switch to a list concept, calling them 'elements.'" Well, as far as "one-dimensional" arrays are concerned, JDan's "list" concept is probably about as good as it gets—and don't bother declaring these with dual-index syntax even though you could. Why make things more complicated than they have to be? Only make arrays with two or more "dimensions" when you need them for separating different types of data from each other. So that's an easy decision. The trouble comes when you actually do need a "multi-dimensional" array. What then?

With 2D and 3D arrays, the plane-row-column model is good to use because it's ubiquitous across programming languages. Because it is thought of in terms of rows and columns, a two-dimensional array is called a *table* or *matrix*, the first dimension being the row number, the last being the column number. Here's an illustration of how this plays out. In a unidimensional array, we'd access the second datum in the list with a simple, direct index of **`[2]`**. However, once we have multiple lists, we have to do the row thing. If we create **`arArray[4,4]`**, then item **`arArray[2;3]`** is the element in the spot that we're thinking of as being "second row, third column":

	Column 1	Column 2	Column 3	Column 4
Row 1				
Row 2			**2;3**	
Row 3				
Row 4				

This 2D array has 4 rows and 4 columns, but you needn't make the dimensions of 2D arrays (or any multiD arrays) evenly matched. For example, **`declare arArray[5;7]`** creates an array with 5 elements in the first dimension (5 rows) and 7 elements in the second dimension (7 columns) for 35 total elements.

As with unidimensional arrays, you can declare two-dimensional arrays with a constant—but if you do so, make sure to include a semicolon or comma between each array element, including the higher dimensions. For example, this syntax...

```
arArray[] = { {1;2;3} ; {4;5;6} ; {7;8;9} ; {10;11;12} }
```

...will create a **`[4;3]`** array. Notice that there are 4 subgroups of 3 elements each? Each subgrouping specifies one row, so you can visualize the elements as stacking up like so:

1	2	3
4	5	6
7	8	9
10	11	12

You can think of a three-dimensional array like a polygon of some kind, with "planes" which are essentially different tables (plastered like posters on the sides of the polygon), the tables having rows and columns (each one will have the same number of rows/columns). For example, **`declare arArray[5;2;4]`** creates an array with 5 tables, each of which has 2 rows and 4 columns (for a total of 40 elements).

By the way, constants work for declaring arrays with three (and more!) dimensions, but the proper nesting of the braces (the **`{`** and **`}`** symbols) can get hard to read, let alone construct:

```
arArray[] =

{

        { { 111;112;113;114} ; {121;122;123;124} ; {131;132;133;134} } ;

        { {211;212;213;214} ; {221;222;223;224} ; {231;232;233;234} }
}
```

This will create a **`[2;3;4]`** array—two main groups (tables, signalled by the brown braces) that have three subgroups (rows, each enclosed by green braces) of four elements (columns, each in purple). When you declare an array like this, each group and subgroup (or sub-subgroup) must be the same size as the other groups, subgroups, or sub-subgroups that it matches up to, or you will get an error.

So there you have it: the plane-row-column model. But as far as a common manner of conceptualizing arrays is concerned, that's it. Cognition-wise, it's open season on arrays with four or more dimensions. "I'm not aware of any consistent terminology, since most of us live in a three-dimensional world," says JDan. "The concept of a 4th dimension is difficult to imagine. I usually then start adopting notions like drawers, cabinets,

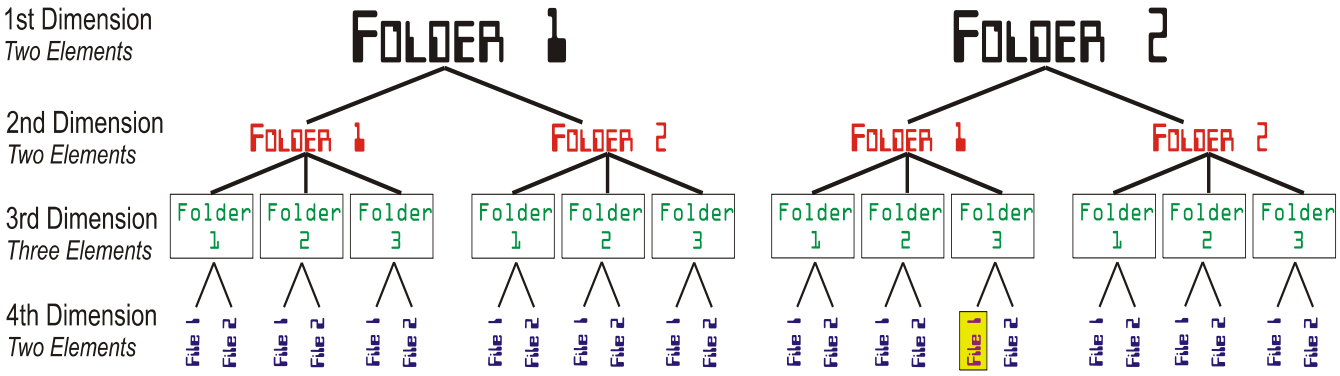
In Memoriam

The macro language allows up to 10 dimensions in an array, and each dimension can have up to 32767 elements. However, even though the macro system supports these potentially huge arrays, you are physically limited by the amount of memory you have. If you try to create a huge array, and there is not enough memory, then the macro system will abort your macro with an error. So in reality you can't get anywhere near the maximum that the macro system allows. In fact, a two-dimensional array of **`[32767;32767]`** would require 12 GB of memory (three times more than current 32-bit Intel/AMD processors will even support). True, 64-bit processors are now available, but they'll take awhile to become common...

rooms, etc." If rooms and cabinets seem a bit far-fetched for computer programming, however, there is another option. Programmers apparently don't use it (none of them told me about it, at any rate—I had to figure it out on my own) but I certainly find it helpful because it's already quite familiar to most of us. Not only that, but it works not just for more complicated structures but even for "one-dimensional" arrays: introducing—dum-da-da-dum!—the COMMAND-LINE FILE ANALOGY!

Yes, the simple, underrated file. Why *not* think of an array as a computer that contains files (array elements) in folders (all index ranges before the rightmost)? Heck, all dimensions in any array are simply funny-money organizational heuristics anyhow, doing for array values what logical drives and directories do for computer files. If you really want to see how a computer keeps track of files, snoop around the hard disk with a hex editor. The CPU/Storage team pays little attention to your carefully organized folders, instead employing a method much like the PerfectScript array data-retrieval formula. And since what goes on under the hood with PerfectScript arrays and hard-disk data is similar, why not employ the same analogy to establish a "user interface" for them both?

From this perspective, a unidimensional array would be like putting all your files in the root directory—there are no folders or subfolders, so to open a file, you never have to type pathnames at the command prompt. Once you start organizing things into folder hierarchies, you have to use pathnames to get to the files; once you start creating a multiD array, you have to use "pathnames" to get to the array elements. When you work with files using a command-line interface, you're getting to your files through a series of hierarchically arranged folders. You simply type out the pathname in the form of **topfolder/nextfolder etc.../bottomfolder/filename**. With a multiD array, you do much the same thing. In the same way that you can lay out your folder hierarchy in a chart (as Windows Explorer does), you could also chart an array hierarchy. The main difference is that files can reside in any folder on a hard drive, but in an array, the files can only reside in the bottom-most folders, those at the bottom of the hierarchy. For example, you might chart a **`[2;2;3;2]`** array like this...



...after which you can just follow the chart whenever you need the index number of any given element. Let's say you want your macro to use the item in the **yellow box**. No sweat; just start at the top and follow the chart downward, concatenating the numbers as you go along. The first tier (top level) gives you 2; the second tier gives you 1; the third, 3; the fourth, 1, so you end up with element **`arArray[2,1,3,1]`**. It's actually very much like going back to the old DOS days when you had to manipulate computer files with a command-line interface rather than a GUI—to

perform an action on a file, you had to enter the whole path at the command prompt just so you could get there. In **arArray[2,1,3,1]**, that last **1** is really the only thing that you want. That's where the little nugget of data is. But you can't retrieve that nugget unless you give the macro the relevant "path."

Reading Dimensions

It's time to get the most basic newbie-gotcha out of the way (this one tripped me up for quite a while before I got used to it). Array dimensions are talked about in left-to-right terms, the same way we read. For example, in a six-dimensional array such as **arArray[5;4;8;7;9;2]**, **5** is the first dimension, **4** is the second, **8** is the third, and so on until **2**, the last, sixth dimension. So you start at the left and count upward toward the right. Seems straightforward enough, eh?

However, this conventional way of discussing array dimensions raises a logical problem. Well, maybe it's not technically a logical problem, but an amateur, non-mathematical programming novice like me might well be logically tripped up by the dimensionality issue. **arArray[2]** is a one-dimensional array, right? And **arArray[9;2]** is a 2D array. **arArray[7;9;2]** is a 3D array, **arArray[8;7;9;2]** is a 4D array, **arArray[4;8;7;9;2]** is a 5D array, and all we do to get our 6D array, **arArray[5;4;8;7;9;2]**, is add the **5** on at the end. Notice my terminology: the last dimension we added is at the *end*. It's the last step in the process to get a 6D array. The previous dimensions remain the same as they always were, unchanged since we first typed them in. So shouldn't **5** be thought of as the *last* index range, the *sixth* dimension, rather than the first one?

The same logic applies going the other way (reducing dimensions instead of adding them).

The rightmost dimension in an array is always the most basic index range—the one that should be thought of as holding the actual data if you use the "command-line file analogy." So in **arArray[5;4;8;7;9;2]**, the rightmost dimension is the most basic one (columns, if you use the geometrical metaphor); the adjacent dimension on the left is the next most basic (rows), and so on.

This poses a problem. It's logical to think that the "first dimension" would be the rightmost (the one with the index range of "2"); the "second dimension" would be the one with the index range of "9"; the "third dimension" would be the one with the index range of "7"; and so on—not the other way around. This would make array terminology more logically consistent. Under the prevailing system, **2** is the sixth dimension in **arArray[5;4;8;7;9;2]**, but if you lop off the leftmost dimension to get **arArray[4;8;7;9;2]**, all of a sudden **2** is the fifth dimension instead of the sixth. Sure, there are four less sets of **2**, but in neither case has it ceased being the most important, fundamental dimension; surely it would make more sense to think of it as the first dimension, **9** as the second, **7** as the third, and so on, instead of getting a new status each time the leftmost dimension is removed. After all, the array as a whole might have fewer dimensions, but the dimensions that remain haven't changed!

In addition, the geometric terms we use for the three rightmost dimensions (planes, rows, columns) imply that columns are the first dimension, rows are the second dimension, and planes are the third dimension. It is therefore natural for neophyte programmers to think of columns as always being the first dimension, rows always being the second, planes always being the third, etc., etc. Yet when programmers start talking about a real live three-dimensional array, it's the exact opposite! The columns are spoken of as the third dimension while the planes are the first.

JDan says that it's done this way "because we are left-to-right readers"—but, in fact, not all of us are. What if your native language is Arabic? Oh, well. In their infinite wisdom, the Primal Programmers have done what they have done, however little sense it may seem to make. From now into infinity, array dimensions will be discussed in left-to-right terms, so get used to it.

`Dimensions()` and ***`array[]`***

Now that we know what array "dimensions" really are and have various options for conceptualizing them, we're ready to examine two directly relevant PerfectScript features: the command **`Dimensions(`*`VariableName;IndexOption`*`)`** and a special reserved operator for arrays, ***`array`*`[0;`*`IndexOption`*`]`**. The former dates from time immemorial, while the latter has been around in its present form only since WP7. However, we'll look at the first only in passing; the second—which is nice and compact—performs the same tasks with less space, so it doesn't take rocket science to decide which we ought to use. The reserved array operator ***`array`*`[0;`*`IndexOption`*`]`** requires you to use the name of your array in place of ***`array`*** (that's why "***`array`***" is italicized) but you probably already know it as **`array[0]`**—with only one parameter—because that's how the PerfectScript manual describes it. When appearing in this familiar form, the array operator holds the total count of elements in the specified array. Again, that's *all* the elements in an array, not just the index range of a specified dimension or whatever. For example, **`arArray[2;3]`** has 6 elements, so using **`arArray[0]`** will return a value of "6."

Now You See It, Now You Don't

There, I knew you'd remember it. The bad part is that the macro manual offers no way to find out about ***`array`*`[0;`*`IndexOption`*`]`**'s very important second parameter, the ***`IndexOption`***. The first param, **`0`**, is not optional and is always the same because its function is merely to signal PerfectScript that this item isn't an array (by definition no PerfectScript array can have "0" as an index range, nor can any array reference have "0" as an index, so an item with brackets and a "0" has to be the array operator). That's fine—yay for the first parameter—but the documentation fails to mention the second param, and since it's optional, when you leave it off you don't get smacked with the kind of syntax error which might otherwise serve to let you know that there's some unaccounted-for factor in your macro's mix. Luckily, there is documentation on **`Dimensions(`*`VariableName; IndexOption`*`)`**, the command that ***`array`*`[0; IndexOption]`** supersedes, and the command acts almost exactly the same as the operator. Understand one, understand the other.

`Dimensions()` Reveals All

Of the two parameters in **`Dimensions(`*`VariableName; IndexOption`*`)`**, the first is the array name. The second is the cool one. This param, ***`IndexOption`***, specifies the kind of information you want about that array. There are four enumerations:

`ElementCount!` (numeric equivalent **`0`**)

This is the one we're already familiar with in form of its numeric equivalent, **`0`**. If either **`ElementCount!`** or **`0`** is passed, then the total number of array elements is returned.

`DimensionCount!` (numeric equivalent **`-1`**)

Things get more interesting with **`DimensionCount!`**, which tells you how many dimensions the array has.

`IndexLimit`*`x`*`!` (numeric equivalent ***`x`***)

Most impressive is this highly flexible last enum, where ***`x`*** is a number from one to ten that refers to a dimension according to its place in line (first, second, third, and so on; for more information, see "Reading Dimensions" above). **`IndexLimit`*`x`*`!`** returns the index range for the specific dimension denoted by ***`x`***. Of course, to use this enum, you have to know or find out how many dimensions the array has. (You can't command PerfectScript to return the index range of a dimension unless you tell it which dimension you're after—i.e., whether that dimension is first, second, third, or whatever; PerfectScript can't read minds!) You can also simply use a standalone digit from **`1`** to **`10`** to pass this enum. For example, after doing **`declare arArray[2;8;6;1]`**, you could use **`Dimensions(arArray[];3)`** to get the index range of **`arArray[]`**'s third dimension (so the command's return value would be 8).

Finally, the razzle-dazzle second param of **`Dimensions(`*`VariableName; IndexOption`*`)`** that we've been rhapsodizing about is optional, and if you leave it unspecified, PerfectScript steps in and supplies **`DimensionCount!`** (i.e. **`-1`**) in default, thus giving you the number of dimensions in the array. **Note:** *you can't use the verbal enumerations for the second parameter in WP7, just the numerical equivalents.*

The Power of ***`IndexOption`***

The second parameter of ***`array`*`[0;`*`IndexOption`*`]`** accepts the same values (numeric only, however, not enumerations) that the **`Dimensions()`** command accepts as its second parameter:

- ***`array`*`[0;0]`** returns the total element coun t (and is the default behavior if the second, optional parameter is left unspecified—i.e. if you just type "arArray[0]").

- ***array*[0;-1]** will return the dimension count (how many dimensions the array has).

- ***array*[0;*x*]** will return the declared index range for the dimension specified by ***x*** (the second slot in the brackets). Since arrays can only have from one to ten dimensions, ***x*** has to be a number from 1 to 10; ***array*[0;1]** will return the index range of the array's first dimension, ***array*[0;2]** will return the index range of the array's second dimension, and so on up to 10 (assuming that the array truly has the corresponding number of dimensions—if it doesn't, you'll cause a syntax error).

So just keep that first index at **0** to signal PerfectScript that this isn't an array reference but rather the array operator—and whale away with the second index!

The only difference between **Dimensions(*VariableName*)** and ***array*[0;*IndexOption*]** is what happens when the second parameter remains unspecified. Given an array called **arArray[]**, the command **Dimensions(arArray[])** will return the number of dimensions the array has. In other words, if you leave the second parameter blank, **Dimensions()** defaults to **DimensionCount!** (or **-1**). Not so with ***array*[0;*IndexOption*]**. Instead, **arArray[0]** will be fed a default second param of **ElementCount!** (i.e. **0**), returning the total number of array elements rather than the number of dimensions. (WP DOS 6's ***array*[0;*IndexOption*]** returned the element count, and Corel kept it that way in WPWin for backward compatibility.)

Slices

Now let's talk about "slices," the final "dimension"-related feature of arrays. A slice of an array is a *group* of array elements—a bigger-than-normal piece of the array, starting at some element and extending to some other element. Slices are indicated in brackets like other array element identifiers, but with a mini-ellipsis (two dots) between the two array elements that serve as the starting and ending values. So if you declared **arArray[9]** and then wanted to reference elements four through eight, you'd type "**arArray[4..8]**."

Keep 'em Together

Make very sure that you don't put a space between the two periods that constitute slice ellipses. Type ".." rather than ". ." or your macro probably won't compile. So keep those periods together.

Slicing off Arguments

What's potentially confusing about array slices is that unlike normal references (i.e. when you're targeting single elements of an array), the arguments in a slice *are* optional; when carving slices

out of **arArray[9]**, you could have **[..8]**, **[4..]**, or even **[..]**. Here are the relevant rules PerfectScript goes by when it sees that slice arguments have been left off:

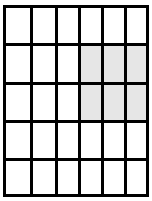
> **1.** If you don't specify the starting element, the macro will start the slice at the very beginning of the relevant dimension. Thus, given **arArray[9]**, **[..8]** is the same thing as **[1..8]**.
>
> **2.** If you don't specify the ending element, the macro will stop the slice at the very end of the slice's dimension. Thus, given **arArray[9]**, **[4..]**is the same thing as **[4..9]**.

Given these rules, you can see that with **arArray[9]**, our example array, **[..]** is the same thing as **[1..9]**. (Note that there's one more rule, but comes into play only with multidimensional slices, so we'll lay a bit more groundwork before dealing with it).

Slicing Contiguous Elements

Elements in a slice must be adjacent to each other—you must specify a continuous range of elements for each dimension. In a unidimensional array, this is so easy that it's next to impossible to mess up. For example, an array such as **arArray [9]** could support various slices: **[1..3]**, **[4..8]**, **[2..5]**, **[..]**, or whatever. In regular English text, these would be written "1-3," "4-8," "2-5," and "1-9" respectively. As long as you don't make any obvious howlers—as long as you make sure that the first number is lower than the second and that you don't list a higher number than the array itself has—you'll have a syntactically correct slice because the embrace of each range extends to all numbers between the starting and ending digits. For example, **[1..3]** includes 2. You can't exclude 2; there's simply no PerfectScript mechanism for doing that. Likewise, **[4..8]** includes 5, 6, and 7. You get the picture. Unless you do something like **[2..1]**, you won't get a compile-time syntax error. (Of course, if you don't really know what you're doing and you type something like **[1..9]**, you may well be including more array elements than you want, thus writing a semantic error into the macro—but that's a different story.)

The syntax of multidimensional arrays gets trickier because you have to slice across both rows and columns. Let's declare a two-dimensional array with 5 rows and 6 columns, **arArray [5;6]**. To specify a 2-row, 3-column slice, you'd use **arArray[2..3;4..6]**. In this slice, the first range ("2..3") indicates the rows that will be affected (rows two and three). The second range ("4..6") specifies the columns that will be targeted (four, five, and six).



A 2-row, 3-column slice

If you want to think of **arArray[2..3;4..6]** in terms of the command-line file analogy, the first range indicates the "folders" involved (folders 2 and 3), while the second range specifies the particular "files" you're after in each folder (files 4, 5, and 6). Be aware that the analogy doesn't completely reflect reality, though. When getting regular computer files out of different folders, you can retrieve an asymmetrical number of files with wildly different names: from folder 2 you could get two files, A.txt and B.wpd, while from folder 3 you could get four files: Whompdiddly.html, XYZ.pdf, Diddlybop.gif, and Gizzard.meat. But if

you're slicing a multiD array, you'll always specify a *symmetrical* number of elements (i.e. the exact same number of "files" from each "folder"), and the elements you're after will always have the same names (in the present example, "4," "5," and "6") even though they're in different "folders."

This principle leads us to the final "syntax rule" referred to above (in "Slicing off Arguments") which I said we'd deal with later, and now here we are: if you refer to a multiD array in slice syntax for one of the dimensions, any dimension not syntactically sliced will still be treated as a slice, though just for the element indicated by the index you specify. This is highly convenient. After all, there will be times when you'll want to type **`arArray[2..3;6]`** as an array reference because you want the two pieces of data (the "files" or "columns") at the **`6`** spot in the last index (one piece of data from row 2 and one piece of data from row 3), but you *don't* want any other array elements. You don't want a *group* of elements in that second index—you don't want **`4..6`** but rather just **`6`**.

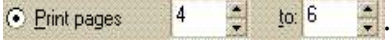
Unhappily, this convenient rule wasn't implemented until WP9, so there's a problem faced by macro writers using WP8 and prior versions, or by anyone who wants to write a backward-compatible macro. If you use WP9 or later and you're writing macros only for yourself and others with these newer versions, you can ignore the next section. The rest of us will pause to look at this issue and its workaround.

The Limitation with WP8 and Earlier Versions

In WP8 and before, the "single-digit, no ellipsis" slice syntax rule doesn't apply. In these versions, once a slice, always a slice: when you slice a multiD array in a given array reference, all the indices in that reference have to be expressed in slice format. So in a given array reference, once you've set up a slice as the first index range, you cannot use regular single-digit indices for subsequent ranges, at least in that particular array reference; *all* the index ranges must be filled with slices. In other words (still working with **`arArray [5;6]`**), you can type something like **`arArray[2..3;6]`** in WP 9 and newer versions; in WP8 or older versions, it has to be something like **`arArray[2..3;4..6]`**.

Well, if the very syntax of slices demands a grouping such as **`4..6`**, or at least **`5..6`**, what's a WP8 (or earlier) macro writer to do when you just want a single element from one of the dimensions? No sweat, really. WordPerfect itself provides a familiar example of how to get around the problem: just treat the column as if it were a single page that you want to print from a long WP document. When the WP print dialog box pops up, you're faced with a column of radio buttons offering various ways to choose the pages you wish to print. The fourth radio button lets you specify a range: 

.

Usually you'll use this feature to print a range of several pages, like four and six, so you type "4" into the first box and "6" into the second: 

. But sometimes you just want to print a single page, not a group of them. In such situations, you can't simply type that page number into the first box and be done with it—you must type it into *both* boxes. So if you wanted to print only page six, you'd put "6" into each box, like so: 

.

This **Print pages** feature works exactly like array slices in which one index represents a single "folder" or "file". If you've already sliced one array argument you must slice them all, but you can still designate a single column (or any other individual dimension) by using the same number to both start and end the slice. In sum, rather than typing **`arArray[2..3;6]`**, users of WP8 or older versions will type **`arArray[2..3;6..6]`**.

WP 8 Bug

There's a slicing bug in WP 8 that can trip up unwary macro writers who want to use slices to extend an array's dimensions (for details on this task, see "Extending Dimensionality," below). JDan explains that when the macro system is instructed to deal with slices, it "collapses" any *uni*dimensional arrays that may have been declared with *multiple* dimensions—i.e. arrays such as **`arArray[1;3]`**. Says JDan, "a [1;3] or [3;1] array is essentially the same as a [3] array internally." Of course, if you declare your array as "[1;3]" instead of just "[3]", *you* still have to refer to it in 2D terms (see above, **Conceptualizing Arrays | Why Bother With All That Theory?**)—but the macro engine doesn't. In fact, in order to work with slices, it can't. Unless it gets rid of any such dummy dimensions, it can't tell whether or not a slice from one array will fit into another."So when the macro system deals with array slices," JDan continues, "it scans the slice for any of these single-element dimensions, and collapses the total dimensions of the array to eliminate them."

In WP 8, unfortunately, there's a bug related to this behavior. If you create a unidimensional array with only a single element—viz. **`declare(arArray[1])`**—and try to use slices, the macro generates an error and shuts down because the macro engine creates the WordPerfect equivalent of a black hole, collapsing the array completely into nothing. Apparently in this particular situation, the WP 8 macro system is coded so that it must collapse *something*, even if the single-element array actually has a value and thus doesn't qualify for collapse because it's not empty. This is a problem if you want to build a one-dimensional array automatically, augmenting its dimension as you go using the techniques discussed below in "Extending Array Dimensions." You would start out by declaring the empty unidimensional array; and that would work fine, but as soon as you tried to augment the array (which requires using slices) PerfectScript would create the black hole and the macro would crash.

JDan eliminated this bug in WP 9. If you use 8, however, there are workarounds. One is simply to use a dummy dimension on purpose: do **`declare(arArray[1;1])`** instead of just **`declare(arArray[1])`**. In this case, the macro engine gets its ya-yas out on the dummy dimension and doesn't feel the need to go on to collapse the second, "real" dimension.. Of course, then you have to continue using the 2D syntax throughout the macro, and JDan notes that "it makes your macro code appear to be more complex by using multiple dimensions. Later on, someone (maybe even you yourself) might wonder why the array has 2 dimensions instead of just one. Understanding and maintaining the macro code becomes more difficult in the future." The other solution is to use **`IF`** statements to see whether you have WP8 or a unidimensional array, and if you do, change the code slightly. For more on this issue, see the end of this document, which refers you to a WPUniverse webforum thread in which JDan deals with it.

Building Arrays: Best Practice

How should you build your arrays? Everyone knows that you can combine several single-dimension arrays into one multidimensional array to improve load time when you need to—for example—populate arrays with data that is used to validate user entries from a dialog. But putting a multiD array together is a time-consuming and resource-intensive operation, and there are various options to choose from. Which is best?

Post-Declaration Direct Assignment to Individual Elements

For a neophyte, the easiest way to get data into an array is by working through the process step–by-step, first declaring the array and then assigning values to each array element individually. Let's say you're a tailor building a database of male customers according to their waist size, and you want to both lump them into large categories (Small, Medium, etc.) and include their exact size. You'd need a 2D array; your top-level dimension, the "first" one, would contain one "folder" for each customer, while your bottom-level, most fundamental dimension, the "second," would contain three elements: each customer's name, general size category, and exact size. So if you have four customers, your first dimension will be 4, and no matter how many customers you have, your second dimension will be 3:

```
declare arArray[4;3]
arArray[1; 1] = "John"
arArray[1; 2] = "M"
arArray[1; 3] = 44
arArray[2; 1] = "Mike"
arArray[2; 2] = "S"
arArray[2; 3] = 32
arArray[3; 1] = "Frank"
arArray[3; 2] = "S"
arArray[3; 3] = 19
arArray[4; 1] = "Ron"
arArray[4; 2] = "M"
arArray[4; 3] = 35
```

Note how string data-types are enclosed in quotes, but numerical data is not.

For Loops

If you have a lot of data, the first method gets pretty cumbersome to type out in code. With a little more experience, programmers often employ a **`For`** loop of some sort instead. In the example below, featuring **`ForEach`**, there are three loops instead of just one. Why? Remember, the last index in the array declaration, the "column" spot, denotes the real array elements. Now recall that the array in question has three columns, and since those columns are where the actual

data goes, we have to assign the values to the array's columns (as opposed to the rows). Each loop thus contains all the values for one of the columns (even though they actually look like rows in the code).

In a **ForEach** loop there are only 2 parameters, the **ControlVariable** and a list of items to put into it (for example, in the first loop below, the list is **{"John"; "Mike"; "Frank"; "Ron"}**, each of which successively gets placed in the variable **n**). The loop iterates as many times as there are list items, and each time it goes 'round it jumps to the next list item, so by the time the loop stops, each list item has been successively placed into **ControlVariable** and has therefore passed through the loop's statement-block meat grinder. As far the array indices are concerned, the first loop populates **[1;1]**, **[2;1]**, **[3;1]**, and **[4;1]**; the second populates **[1;2]**, **[2;2]**, **[3;2]**, and **[4;2]**; and the third loop finishes up with **[1;3]**, **[2;3]**, **[3;3]**, and **[4;3]**:

```
declare arArray[4; 3]

count = 1
ForEach (n; { "John" ; "Mike" ; "Frank" ; "Ron" } )
        arArray[ count ; 1 ] = n
        count = count + 1
EndFor

count = 1
ForEach ( n ; { "M" ; "S" ; "S" ; "M" } )
        arArray[ count ; 2 ] = n
        count = count + 1
EndFor

count = 1
ForEach ( n ; { 44 ; 32 ; 19 ; 35 } )
        arArray[ count ; 3 ] = n
        count = count + 1
EndFor
```

The importance of the columns is made clear by the fact that each loop is dedicated to one of them. But rows also become important once we move *inside* the loops; after all, we do need to tag each piece of data with its appropriate row somehow. The table-coordinate system won't work unless each array element is attached to both a column *and* a row, and we use the loops' increment value to take care of the row part. Here's the first line of the first loop's statement block:

```
arArray[ count ; 1 ] = n
```

Let's examine how it works. First, keep in mind a few preliminary items.

- **`arArray[count; 1]`**, of course denotes an array element (now that the array has been declared, it can't be anything else). Its last index, **`1`**, denotes the column that is supposed to be populated with the **`ForEach`** loop's data.

- Equally important in **`arArray[count; 1]`** is the first index, **`count`**. This index denotes the particular row number with which an array element will be tagged—the first of the two "table" coordinates that PerfectScript uses to target an array element.

OK, now let's plunge into the process:

1. The first time through, the **`ControlVariable`**, **`n`**, is set to the first value in the loop's data list (the string datatype value **`"John"`**).

2. The var **`count`** was set to **`1`** immediately before the loop fired up, so **`arArray[count; 1]`** is actually **`arArray[1; 1]`** at this point.

3. The increment value, **`n`**, is passed to the array element which is currently (still in the first iteration) **`arArray[1; 1]`**. As a result, the very first array element, the first datum of column one, now has a value (**`"John"`**).

4. The first line of the statement block is finished, so PerfectScript jumps down to the second line:

```
count = count + 1
```

5. On the right side of the equals sign, PerfectScript adds 1 to **`count`**, which is also currently **`1`**. In other words, the macro engine performs 1+1. The result of the expression is **`2`**.

6. PerfectScript passes this result to **`count`** on the left side of the equals sign, overwriting its previous value of **`1`** with the new value of **`2`**.

7. The loop hits the **`EndFor`** and pops back up to the top; it has to iterate four times because there are four items in its second (data-list) param.

8. Now the next item in the data list is passed to **`n`**, so **`n`** now contains **`"Mike"`**. Since the var **`count`** is now **`2`**, **`arArray[count; 1]`** is now actually **`arArray[2; 1]`**. Thus, when the first line in the statement block is finished, the second datum of column one has been assigned the value **`"Mike"`**.

9. In the second line of the statement block, **`count`** will again get bumped up by one. Then the loop will iterate twice more, and by the end of the whole process, the first column of the three-column array will have been completely populated. All four of its slots will be filled, each one containing a name from the loop's data list. The same thing

will happen in the other two loops, where columns **2** and **3** will get populated with the lastname initials and the peoples' ages.

"Lump-sum" Direct Assignment

While a **ForEach** loop might seem pretty efficient compared to the other For loops, it is still slower than a direct assignment would be. Direct assignments elide the need to declare an array; if you populate the array in one fell swoop of an operation, there's no need for the **declare** syntax (it's just like a normal variable to which you assign a value upon its first appearance without using **declare**). This omission, and the fact that there's only a single operation involved, makes "lump-sum" direct assignment the fastest, least resource-intensive way to populate arrays. In sum (pun intended), it would be a good idea to replace the loops with a single assignment statement like this:

```
arArray[ ] =
{
        { "John"  ;  "M"  ; 44 } ;
        { "Mike"  ;  "S"  ; 32 } ;
        { "Frank" ;  "S"  ; 19 } ;
        { "Ron"   ;  "M"  ; 35 }
}
```

There is a problem, however. This streamlined, petite princess of a method is accompanied by an evil stepmother, an inherent limitation in the macro system. PerfectScript looks at the assignment statement (on the right of the equals sign) first and builds a temporary array out of it. Only then, when the assignment can be made *en masse*, does PerfectScript take the data, cross over to the left side of the equals sign, and dump it into **arArray[]**. That may not seem like a big deal, especially if you've got a newer computer with oodles of memory. But RAM isn't the limitation here. The problem arises because making the temparray requires the macro engine to push each of the array elements onto the internal macro data stack—and the internal macro data stack is limited to 250 items. So if the array will have more that 250 total elements, you're out of luck. You'll get an error message and a nonworking macro.

Prebuilt Columns

If the direct-assignment limitation disallows direct assignment for your array, a good alternative is build the array in pieces, making individual arrays first and then combining them. You can do this either by row or column. We'll take columns first. Build each column as a separate array, then assign these unidimensional arrays into the final multiD array:

1. First, make the columns as separate unidimensional arrays. Use the "lump-sum" direct-assignment method:

```
arSubarrayA[ ] = { "John" ; "Mike" ; "Frank" ; "Ron" }
arSubarrayB[ ] = { "M" ; "S" ; "S" ; "M" }
arSubarrayC[ ] = { 44 ; 32 ; 19 ; 35 }
```

2. Next, declare the multiD array that you really want, using the **`array[0]`** syntax to give it one of the temporary array's total element-counts as a row index. A unidimensional array isn't usually thought of as having rows, but it can be conceived that way; each element can be considered as constituting a row, as the table to the right illustrates. So when you give your multiD array a row index equal to the total number of elements from a unidimensional array, you give it the same amount of "rows" that the unidimensional array has. Since the number of rows will stay the same from the unidimensional arrays to the multiD array, we can use the unidimensional row count (equal to the total elements) as a highly convenient way of specifying the multiD row index without hard-coding it:

row 1	**arSubarrayA[1]** ("John")
row 2	**arSubarrayA[2]** ("Mike")
row 3	**arSubarrayA[3]** ("Frank")
row 4	**arSubarrayA[4]** ("Ron")

Rows in a unidimensional array

```
declare arArray[ arSubarrayA [ 0 ] ; 3 ]
```

3. Finally, populate your new multiD array with the old temporary ones. In order to assign each temporary array in its entirety to a *single* column in the new array, you have to use slices (see above, last section of "Dimensionality Demasked"). In the following three assignments, the multiD array's row index is simply left as it already stands, but you can't do that with the column index; you have to isolate the columns so that the temporary array elements go into the column you want, not just whichever of them PerfectScript defaults to. Designating each column as an individual slice allows us to do this:

```
arArray[ .. ; 1..1 ] = arSubarrayA[ ]
arArray[ .. ; 2..2 ] = arSubarrayB[ ]
arArray[ .. ; 3..3 ] = arSubarrayC[ ]
```

Obviously we couldn't make these assignments with something like **`arArray[1,1] = arSubarrayA[]`**, since we'd be trying to cram an entire array into one single datum. The macro would generate a syntax error, refusing to compile. Most people wouldn't make such a mistake, but there is a more subtle gotcha waiting in the wings. Since we're passing entire arrays into another entire array, it might seem logical to have coded these three assignments as **`arArray[..; ..] = arSubarrayA[]`** or even just **`arArray[] = arSubarrayA[]`**, etc. However, we wouldn't have gotten the results we wanted. Remember, we're passing a unidimensional array here—that's only one single column, and it's going into a *three*-column array. When given one of the unidimensional arrays, how is the macro system to know which of the three multiD columns that the uniD array is supposed to go into? Well, if we don't tell it which column we want (and slices are the only way to differentiate them), PerfectScript will probably just default to the first column. So our code won't work as we intended, but it *will* execute exactly as we wrote it:

A. **`arSubarrayA[]`** goes into **`arArray[..;1..1]`**.

B. Next, **`arSubarrayB[]`** also gets passed into **`arArray[..;1..1]`**, thus overwriting the data from **`arSubarrayA[]`**.

C. Finally, **`arSubarrayC[]`** overwrites that in turn.

In other words, when the dust settles, **`arArray[]`**'s *first* column will contain **`arSubarrayC[]`**, the data we had intended for the *last* column. Oops. Meanwhile, **`arArray[]`**'s other two columns won't contain anything, never having been touched by our faulty code. So use those slices!

The final working code looks like this:

```
arSubarrayA[ ] = { "John" ; "Mike" ; "Frank" ; "Ron" }
arSubarrayB[ ] = { "M" ; "S" ; "S" ; "M" }
arSubarrayC[ ] = { 44 ; 32 ; 19 ; 35 }

declare arArray[ arSubarrayA[ 0 ] ; 3 ]

arArray[ .. ; 1..1 ] = arSubarrayA[ ]
arArray[ .. ; 2..2 ] = arSubarrayB[ ]
arArray[ .. ; 3..3 ] = arSubarrayC[ ]
```

Prebuilding with Direct Assignment

Sharp-eyed readers may have looked at the subarrays, which are constructed with the direct-assignment method, and wondered why I didn't just eliminate the subarray step altogether and populate the array slices by direct assignment. Well, I could have, and in fact would have, except I think that using single declared arrays as a mediating step helps illustrate this technique more clearly. But now that it's been illustrated, by all means get rid of the middleman and just do direct assignments:

```
declare arArray[ 4 ; 3 ]

arArray[ .. ; 1..1 ] = { "John" ; "Mike" ; "Frank" ; "Ron" }
arArray[ .. ; 2..2 ] = { "M" ; "S" ; "S" ; "M" }
arArray[ .. ; 3..3 ] = { 44 ; 32 ; 19 ; 35 }
```

Prebuilt Rows

You can build each *row* as a separate array (with three columns apiece, each column containing only a single data element), then assign them to the bigger array you wish to build. Once again you assign the total element-count from a temporary array to one of the multiD array's indices,

and once again you use slices for differentiating between the multiD array's dimensions. In sum, you repeat the treatment of index and dimension required by the prebuilt-column method, just reversing it:

```
declare arArray[ 4 ; 3 ]

arArray[ 1..1 ; .. ] = {    "John"    ;  "M"    ;  44    }
arArray[ 2..2 ; .. ] = {    "Mike"    ;  "S"    ;  32    }
arArray[ 3..3 ; .. ] = {    "Frank"   ;  "S"    ;  19    }
arArray[ 4..4 ; .. ] = {    "Ron"     ;  "M"    ;  35    }
```

Which To Choose?

Here's a comparison of how the various array-building techniques stack up against each other in terms of efficiency—how much time they take and how much space the resulting array occupies in memory.

Array-Building Technique	Code Amount[1]	Seconds[2]
Direct Assignment ("Lump-sum")	186	2.7
Prebuilt Columns	427	4.8
Direct Assignment (Individual)	464	4.1
Prebuilt Rows	517	5.4
For Loop	1523	20.5

[1]Amount of compiled macro object code (not the same as the macro source code that you type or that's generated by the macro recorder).

[2]To generate results above 0 seconds, each procedure was consecutively repeated 1000 times.

Test performed by J. Dan Broadhead, former PerfectScript developer, who used the array featured in the examples just given. The various techniques are listed here in order of most efficient to least efficient. Larger arrays might yield slightly different results.

Most coders will probably want to save their system resources. If that's your overriding goal, here's what the table shows concerning each method:

- The "lump-sum" direct assignment method is by far the most efficient manner in which to build an array, but of course there's always that 250-element limitation. It's a significant problem: obviously the more complex your arrays, the more elements you're likely to have and the more your array-building needs to be

streamlined—and, frustratingly, the less likely you'll be able to use the lump-sum technique.

- The second-best option is to prebuild your columns (the fourth technique demonstrated above), but even this produces 2.3 times the object code of the lump-sum method and takes 1.5 times as long to run.

- Third most efficient is the method we looked at first, that of declaring the array and then assigning values directly to individual array elements. Although this looks a lot bulkier than the prebuilt-column source code, the size difference in compiled object code is negligible; doing it this way produces only 0.2 times the object code as the prebuilt-column way, for a total of 2.5 times more than that produced by lump-summing. It takes 1.8 times as long to run.

- Fourth on the list is the prebuilt-row technique, which is generates 2.8 times more object code than the lump-sum technique and takes 2 times as long to run.

- Last in line is the **`ForEach`** loop, the clear loser, spitting out a whopping 8.2 times as much object code as lump-summing does and taking 7.6 times as long to run.

Some of these results might appear odd because the sizes of the code examples we've seen don't necessarily correspond with the results in the "Code Amount" column. Remember that the source code is not what PerfectScript actually executes when you *run* a macro. You already know that all macros must be compiled before they'll run, but you may not be aware that there is no direct relationship between the quantity of macro source code (the code you type when you create a macro)and the quantity of low-level macro object code produced by the compiler (the code executed by the macro system at runtime). Some macro source statements produce very little macro object code, while others produce a lot of object code.

Remember, too, that you shouldn't necessarily go for the method that produces the most efficient object code; there are other macro-related tasks to consider, such as revising the macro in the future. Consider the size and complexity of your macro before making a decision. If it's not a very big macro and conserving system resources at runtime isn't a big worry, then maybe you should choose **`For`** loops if you find them easier to read and tinker with.

Extending Array Dimensions

Once you've constructed and populated an array, you can swap data in and out of it, erasing old values and replacing them with new ones. But what happens when you want to expand an array beyond its present size? What about keeping all the old values and adding the new ones on top? This would obviously make arrays much more flexible, and the good news is that it can be done. The bad news is that it's a much more complicated task because you have to construct a new space for your new value rather than simply using an already-existing space and overwriting the old value inside.

> **Note**
>
> The methods discussed in this section and the following for increasing the size of arrays do not add dimensions; rather, they add elements and values to already-existing dimensions. It is possible to construct code for adding dimensions to arrays, but that task lies outside the scope of this document.

This section of *PerfectScript Array Theory: Beginning to Intermediate* actually passes from the intermediate to the advanced stage. At this point you know pretty much everything about how arrays work; the rest of this document (the majority of its pages, in fact) is devoted to the single, advanced endeavor of resizing and augmenting array dimensions, but does so in a careful, step-by-step manner that should take you from point A to point B instead of zipping off to point B by itself, leaving you behind. Some folks will probably even find it too careful, too repetitive; more power to them. They probably don't need the material. It's intended for people who are struggling with these concepts, like I did my first time through, and who will appreciate repeated explanations and want to be reminded of important points.

Some readers may already be familiar with a common but unnecessarily kludgy way of doing this:

1. Make a copy of the array.

2. Discard the original array.

3. Redeclare the array with its original upper-index limit + 1.

4. Use a **`For`** or **`ForEach`** loop to assign the original values back over from the copy.

This gets the job done but takes up more time and resources than necessary, for the process can be tightened up with slices (these make it possible to replace the **`For`** loop with a simple assignment). The surrounding issues are fairly complex, however. Once again, JDan comes through with the goods. The following material is derived from his exposition of this topic in the WordPerfect Universe "Macros and Merges" forum. I've changed the names in some of the code examples that follow, but the structure is almost all his.

Extending Unidimensional Arrays

You can extend dimensions on all arrays, no matter how many dimensions they have, but unidimensional arrays are easiest to deal with; in addition, the same basic technique is used no matter how many dimensions your array has. Let's look at it in detail now.

The Basic Extension Technique

In this example, we'll use the special **`array[0]`** syntax to return the upper limit of the elements in an array. This is the same as calling **`Dimensions(arArray[]; IndexLimit1!)`**; it just makes the code a little shorter. Let's assume that we've declared **`arOrigarray[] = {A; B; C; D; E}`** and now we want to add a sixth element with a value of "F." To do this, we'll be dealing with the same two arrays you've traditionally used, the original array and a temporary one, but for the sake of convenience in this document let's call them the "origarray" and the "temparray."

1. First, we make the temparray, declaring it with one element more than the origarray has:

```
declare arTemparray[ arOrigarray[0] + 1 ]
```

2. At this point, the temparray boasts a bunch of empty elements—real useful, eh?—but that's about to change: let's grab the origarray and assign it to a *slice* of the temparray. It'll be a pretty big slice, going from the beginning to the end of the original. Here goes:

```
arTemparray[ .. arOrigarray[0] ] = arOrigarray[ .. ]
```

`arTemparray[]` now has only one empty element left. All other elements in the array's range contain values. (If this syntax doesn't make sense, review **Dimensionality Demasked | Conceptualizing Arrays | `Dimension()` and `array[]`** and **Slices**, above.) **Note:** *In WP8, slices must be used not just on the left side of the "=" sign but also on the right. In WP9+, you can use empty braces on the right.*

3. Now we're cookin'. We've put the origarray's values into the temparray, and because we used slices for that operation, we retained the extra empty element. In other words, we've manipulated the temparray until it's exactly what we're looking for; the only problem is that we need the origarray's name on it. But that's no sweat. The temparray has everything we want, so we can use a straight overwrite:

```
arOrigarray[] = arTemparray[]
```

What PerfectScript actually does on an overwrite, by the way, is to first assess the entire expression without doing anything. Once PerfectScript sees that it's been asked to overwrite the origarray with the temparray's values, it actually begins the process. First, it picks up the origarray's name in one hand (i.e. copies it to memory) but discards the array

itself. Next, PerfectScript picks up (i.e. copies to memory) the temparray. Finally, the macro engine slaps the old array's name onto the copy of the new array, and hey presto! your array's dimensions have been extended.

4. To clean up, we get rid of the temparray:

```
discard arTemparray[]
```

5. At the end, we fill in the new element of the (newly extended) old array—after all, there's no reason to extend a dimension unless we're going to use the new slot:

```
arOrigarray[6] = F
```

So finally, we've got this:

```
arOrigarray[] = {A;B;C;D;E}

declare ar Temparray[ arOrigarray[0] + 1 ]

arTemparray[ 1 .. arOrigarray[0] ] = arOrigarray[ .. ]

arOrigarray[] = arTemparray[]

discard arTemparray[]

arOrigarray[F] = 6
```

Becoming A Functionary

It's great to know the basic technique necessary to add elements to a unidimensional array, but why stop there? Why not make life simpler by reformulating this as a function so we can call it multiple times in the same macro without rewriting the same kind of code?

Let's call our new function **`Extend()`** and give it two params, the origarray and the number of extra elements we want to add:

```
Function Extend (arOrigarray[]; n)
    declare arTemparray[ arOrigarray[0] + n]
    arTemparray[ .. arOrigarray[0] ] = arOrigarray[ .. ]
    arOrigarray[ ] = arTemparray[ ]
    return ( arOrigarray[ ] )
EndFunc
```

We've called the first param **arOrigarray[]** to remind us that it takes an array. (Note also how we included the brackets—you can call a var or param anything you like, but if it happens to be an array, the brackets must always come along for the ride or your macro will generate a syntax error). The second param is also appropriately descriptive (**n** for "number") . The function returns an array with the new size but retaining the old name and contents (there's just some additional empty space in it now). Here's a snippet that uses **Extend()**:

```
arTest[] = {"A"; "B"; "C"; "D"; "E"}   //declare the array

arTest[] = Extend (arTest[ ]; 1)   /* call the function, adding one element; then
                                      overwrite current array arTest[] w/new structure */

arTest[6] = "F"   //put a value into the new element
```

Why Return When You Can Overwrite?

Now, what if you were having a problem with code bloat? Well, you could streamline this function a bit by eliminating the resource-intensive next-to-last step, where the new temporary array is renamed as the old array. Instead of doing this, you can just return the new temporary array. Careful, though. You might be tempted to simply delete ~~**arOrigarray[] = arTemparray[]**~~ and ~~**return (arOrigarray[])**~~, substituting **return (arTemparray[])** for them. But it's not quite so simple.

Let's start from the ground up, theory first. When you pass parameters to a function or procedure in the normal way, what PerfectScript does is make a *copy* of the variable for the routine's internal use; the original that actually exists in the macro remains out there, unmanipulated. This is the procedure by which the macro system insulates local vars in functions and procedures from the rest of the macro; if this mechanism weren't operative, all vars would be global vars. Generally, then, the extra copy is a desirable thing, but it does create a situation in which two of the same variable take up your computer's RAM. For most variables this is no big deal, but arrays can get huge, so multiple instances might well become a resource problem. And with this particular function the issue is made worse because inside **Extend()** we create yet another copy of the array (slightly bigger)!

When **Extend()** returns, the 2 copies used internally (the copy of the original that was passed into the **arOrigarray[]** parameter, and the **arTemparray[]** copy) are discarded. But while **Extend()** runs, there are three versions of the array occupying the macro system's allotted memory—the original array out in the main macro and the *two* copies inside the function. That's not a very efficient use of resources. It'd be great to eliminate at least one of these memory transactions, and we can do just that by forcing the macro system to pass a *reference to* the original array into the function rather than a *copy of* it (for more information, see the next section, "Passing Parameters by Reference"). Once we've done that, we need create only the internal **arTemparray[]**. The extra copy was there to provide the old values to the new temporary **arTemparray[]** array, but a reference could also fulfill that need.

If you're fretting about returning an array with a name that's different than the original because the original name is what's used in the macro's body, don't worry. Remember, after the function runs, we have to *formally* pass the result back to the origarray—which of course is still hanging around outside the function (in the main macro)—or else that result will just sit idle in memory. If we want to do anything with the function's result, it must be formally passed into something else that can be directly manipulated. Well, in our original basic-technique code (before we developed the function) we sensibly passed the result into the first, original, main-macro version of the old array, overwriting that version to avoid the resource problem that would arise if we just left it hanging around in memory. The byproduct of this tactic is that no matter what the name of the array generated by the function, the old array name is automatically retained. So we can just keep right on doing that, or—better yet—we could change this into a procedure, rather than a function, and perform the overwrite *inside* the routine. After all, we're now going to be able to manipulate the origarray directly, and this would allow us to include the overwrite inside the routine, encapsulating yet another step so that it doesn't clutter the main macro.

In sum, we have everything to gain and nothing to lose by proceeding. So what's the secret? We're obviously not discussing default macro-language behavior, so there must be something special we have to do. Luckily, it's very simple, but it's also nonintuitive and easy to mess up, so pay attention. Here goes...

The Divine Miss *&*

If we want to return instead of to overwrite, we need to make three separate adjustments to function **`Extend()`**, but the most important of these comes in the very first line. *The revised function must have an ampersand, **&**, as the first character in the function's first parameter*. The first line of the function will therefore now be **`Procedure Extend (&arArray[]; n)`** (remember, we decided to convert this into a procedure to further streamline the process). Then we continue our adjustments by eliminating the old return code. Now that we're using **&**, we no longer need it:

```
Procedure Extend (&arOrigarray[ ]; n)

    declare arTemparray[ arOrigarray[ 0 ] + n ]

    arTemparray[ .. arOrigarray[ 0 ] ] = arOrigarray[ .. ]

    arOrigarray[ ] := arTemparray[ ]

EndProc
```

Proceeding to Call

So far so good, but here's the nonintuitive part: The ampersand in the procedure itself isn't enough. You must also use one in the procedure's *call*. Yup, every time you call the procedure in the body of the macro, you *also* have to put **&** in front of the original array

name you're using as the first parameter. For example, let's say we want to extend our array by adding two more elements:

```
//declare your array and assign values to its elements:
arTest[] = {"A"; "B"; "C"; "D"; "E"}

Extend (&arTest[ ]; 2)   //call the procedure w/"&", adding two elements...
        /* ...then pass the newly extended array into the original old array, overwriting it
        and thus retaining the original name for the new array */

arTest[6] = F   /* put values into the
arTest[7] = G   two new elements */
```

But hey, remembering to add **&** to the first parameter when calling the function is an excellent tradeoff for the resultant savings in system resources.

Passing Parameters by Reference

Don't be puzzled by this use of **&**. Or rather, do be puzzled by it, since the main reference in the PerfectScript Macro documentation (both online Help and the printable manual) calls **&** a "bitwise operator." Whatever that means and however that works, it has nothing to do with this. In other words, we aren't dealing with the same guy (syntactical context ensures that the two usages remain entirely separate—there's no legitimate coding practice in which one might be confused for the other).

Let us start at the beginning, long, long before little PerfectScript was born, back in the days of legend when mighty Crays shook the ground and Fortran sent light and knowledge sparkling through the circuit-ous cosmos. For a thousand and ten trillion trillion cycles the Primal Programmers administered peace and justice throughout the computing galaxy, allotting mainframe time fairly between various departments, before the dark times—before the Empire. Demigods such as Turing and Minsky then still descended from on high and spoke with humankind, and they taught them the concept of passing parameters into functions. And behold, mortals wrote functions, and they built mighty programs, yea, such as WordPerfect, the secrets of which are now lost and no more, for the Emperor's tyranny has dimmed the capacity to innovate. And in those days they used not one technique for passing parameters into functions, but two: the common method of passing by value, and the more mysterious passage by reference, also called passage by address.

Normally, when you pass a value into a function via a parameter, the macro system makes a copy of the value and lets the function loose on *that* (the copy) rather than on the original value, which remains unchanged. That original value won't be changed or otherwise affected in the routine; any changes made to the parameter are performed not on the original value but on the internal copy. The original hangs out clueless in the main macro—unless, of course, the function's return value is designated for the variable that contains the original, in which case the original value vanishes, replaced according to the function's directives. But this happens only when the function returns its return value to the outside, not *inside* the function itself. Then, finally, the last stage of

the pass-by-value method is the discard: as the function winds things down, it finishes by deleting its internal copy of the original value.

This is all fine and good. It works, it's the default method, and most macro writers remain unaware of alternative ways to pass values into functions. But it's worthwhile getting to know your other option, passing by reference (or address). Some programming languages, says JDan, split hairs and consider passing by reference and passing by address to be different techniques (assigning them differing syntaxes), but they're really the same thing. When using them—or it, rather—you suppress the macro system's usual behavior of making a copy of the value for the function's internal use only. Instead, the function uses and manipulates the original value directly, even updating that value as the function proceeds. In PerfectScript, this is done by making both the function/procedure call (in the macro's body) and the function/procedure definition work together. To specify a reference pass in a function definition, simply preface the parameter's name with "&". But that's not enough (remember, the definition has to cooperate with the call): when the function is called, the variable you give to the reference pass must also be prefaced by "&," as it was in the **`arTest[]`** example above. "When you do this," says JDan, "the parameter in the routine is merely an alias for—not an actual copy of—the real variable that you passed to it when you called it, and when you change the parameter in the routine, you are making changes directly to that variable."

Referential Treatment

Passing by reference is risky—it nullifies the sacred cow of functions (that variables used inside a function only exist internally, so the main macro and the function are insulated from each other). But it's useful anyhow. For one thing, it's basically the same as using the function to receive and change a formally defined global variable in the function, and that's cool because it lets you work around the single-value limitation of functions. Officially, functions are supposed to return only one value, but using either global variables or passage by reference achieves the desired goal of "returning" multiple values because whatever the function does to one of its parameters is directly reflected outward to that value in the larger macro—or, rather, the operation is performed directly *on* that value out in the larger macro because, of course, the parameter isn't a normal internal copy of the original variable, but rather *is* that original variable. So a function technically might not return any values, even the single value it's allowed, but it wouldn't matter because the tasks it performs affect the larger macro immediately. (This opens the door to using a procedure rather than a function in many situations.)

Cool as this is, it's even cooler when passing by reference rather than using global vars. If you use a predefined global variable, the function has zero flexibility; essentially, it doesn't receive that parameter as a variable but rather as a hard-coded value. No matter how many times you run the macro, you can pass nothing but that single global variable into the relevant parameter. In some situations that might be good, but then you'd probably want to use a procedure rather than a function anyway—it's hard to think of a programming problem for which the most elegant solution would be a function that works on a global variable. Passing values by reference, on the other hand, allows you to pass any value you want into a function's parameter as long as you obey the rule that both the parameter (in the function definition) and the variable you're passing into it

(at function call) must be preceded by "&." So if function flexibility and reusability is important, avoid the global var; pass by reference instead.

The other reason to pass by reference is to solve our current problem: the potentially high pressure on memory overhead that occurs when the default passage-by-value method makes that extra internal copy of a parameter's value. If that value happens to be a large array, a system crash could be close behind. But if you pass by reference, the extra internal copy never gets made; the function simply operates on the original value and your memory doesn't suffer from undue stress. However, if your arrays are small, you don't need to worry about this and should probably use a normal function instead of a referential procedure. Because coders are used to having variables inside procedures and functions walled off from the rest of a program, it's easy to forget that a passage by reference suddenly negates the usual scenario—and if you forget, you're likely to have a hard time figuring out why your macro's arrays are all messed up. So while there are certain advantages to the reference technique, think twice before making the pass. The following code examples use passage by reference (until the end of this section with the complete dimension-extension procedure, which I give in both normal and referential versions). But just because the snippets here illustrate passage by reference doesn't mean that's the technique you should use.

And them's the basics of dimension extension, endin' up at function conjunction! Still skeptical? Go ahead and test it yourself. Here's a fully operational test macro that displays a message listing all the elements of the origarray first, then—after you've extended the array—displays a message about the new array element. In a blank WP doc just open the Macro Toolbar (**Tools | Macro | Macro Toolbar**), paste the following code into it, press **Compile & Save**, and run the macro:

```
Procedure Extend (&arOrigarray[]; n)
        declare arTemparray[arOrigarray[0] + n]
        arTemparray[..arOrigarray[0]] = arOrigarray[..]
        arOrigarray[] := arTemparray[]
EndProc

arTest[] = {"A";"B";"C";"D";"E"}

MessageBox (; "Unextended Array"; "array arTest contains " + arTest[0] + " elements.
They are " + arTest[1] + ", " + arTest[2] + ", " + arTest[3] + ", " + arTest[4] + ", and " +
arTest[5] +  ".")

Extend (&arTest[]; 1)
arTest[6] = "F"

MessageBox (; "Newly Extended Array"; "array arTest now contains " + arTest[0] + "
elements. The new element is " + arTest[arTest[0]] + ".")
```

Now let's get fancy and raise the bar, extending the dimensions of *multi*dimensional arrays.

Upgrading to 2D

As is, our function can't extend two-dimensional arrays, just unidimensionals. We can revise our code to redress this shortcoming, but we must first decide how we wish to extend the 2D array. Will we be adding more columns, and leaving the number of rows the same? Will be adding more rows and leaving the columns the same? Or will we be adding more rows and columns?

Adding Columns

Lets take the easiest approach first, which is to just add more columns, leaving the number of rows the same. In this first effort, the only lines changed in the function are the second and third. But these changes are quite complex enough:

```
Procedure ExtendColumns2D ( &arOrigarray[ ] ; n )

    declare arTemparray[arOrigarray[0 ; 1] ; arOrigarray[0 ; 2] + n ]

    arTemparray[ .. arOrigarray[0 ; 1] ;  .. arOrigarray[0 ; 2] ] :=
    arOrigarray[ .. ; .. ]

    arOrigarray[ ] := arTemparray[ ]
EndProc
```

Let's look at lines 2 and 3 in detail.

Line 2: The Right Dimensions

The most obvious change occurs in the **`arArray[0...]`** indices. When using the dimension sysvar to obtain the element count, we can no longer leave off the extra parameter (using the simple ***`array`*`[0]`** form). Now the origarray—that array back in the main macro whose dimension we want to extend—has more than one dimension, so we must explicitly specify all dimensions when manipulating it, or we'll get a syntax error. Previously it was nice and simple; all we had to do in the second line was invoke the origarray's ***`array`*`[0]`** operator (leaving off the second parameter), concatenate that onto the additional number of elements that we wanted, and pass the concatenation into the newly declared temparray. Now, however, we have to use ***`array`*`[0;IndexOption]`**'s second param because **`arOrigarray[]`** will be passed multiple dimensions this time (from the origarray); **`arOrigarray[]`**'s total elements won't sit together all side by side anymore. If we don't specify the "folder" (dimension) that contains the "file" (array element) we want to manipulate, we won't be able to get to that file. (Not only that—remember, once an array's elements are spread out over two or more dimensions, *all* of those dimensions have to be explicitly represented in the code when the array is referenced, so even though we don't care about the rows in this function we still have to "mention" them in our expressions).

Since our particular task at the moment is to extend the array's first dimension, here in the second line we need to declare the new array to have the *same* number of rows as the old array, but a *new* number of columns. In turn, to do this we need to know the original number of both rows and columns; only then can we carry out our planned column extension. But no sweat. Remember your ***array[0;IndexOption]*** syntax? It only returns the total number of array elements if its second parameter remains unspecified. If we specify the second param as a numerical value from 1 to 10, we'll get the number of elements from the designated dimension only. The rows are the top (first) dimension and the columns are the bottom (second), so the numbers we'll need are respectively 1 and 2 (each signaling its respective dimension). The final implementation includes **[0;1]** to get the current number of rows and **[0;2]** to get the current number of columns.

Once that's settled, we can do the column augmentation, concatenating **n** to the index range of the columns dimension:

```
arOrigarray[0 ; 1] ; arOrigarray[0 ; 2] + n
```

Finally, we declare the new array, and the syntactical detective work described above comprises the new array's indices. The first index is designated "the-same-number-rows-in-the-original" and the second index is specified as "the-same-number-columns-in-the-original, plus **n**." So now the new array is the same as the old except for being **n** columns wider. Here's the full line 2:

```
declare arTemparray[ arOrigarray[ 0 ; 1 ] ; arOrigarray[  0 ; 2 ] + n ]
```

Line 3: Transferring the Data

OK, that gives us an **arTemparray[]** array with the new structure we were after (an increased number of columns), but the origarray hasn't been changed; we really haven't done anything useful yet. Line 3 is the place where we'll rectify the situation, putting the old element values into the new array structure. (We'll wait to assign our brand-new values until we get back to the body of the macro.) Since we have a 2D array, we must use the array slice syntax in both dimensions to specify where, in each dimension, we want assign the origarray's data. If everything is sliced correctly, PerfectScript channels the origarray's 2D structure, its rows and columns, into **arTemparray[]**'s corresponding rows and columns. But remember—unless your arrays are exactly the same size in all ways, the process of transferring data from one to the other demands that you tell the macro precisely where to put the data. That's why we have to repeat the ***array[0;IndexOption]*** syntax on line 3 as well as line 2, though here it's on the other side of the expression:, those funny crippled ellipses at the beginning of the slice ("crippled" because there are only two dots instead of the standard three) default to the beginning of the dimension, and the **IndexLimit** numerical value gives PerfectScript an upper limit for the end of the dimension. As a result, you get two complete slices...

```
.. arOrigarray[0 ; 1] ;  .. arOrigarray[0 ; 2]
```

...but because we've been altering **arTemparray[]** rather than our original array, the original hasn't changed. Yes, we used it back in the second line, but that was just to get a number for the

concatenation that was passed to **arTemparray[]**; the main array, **arOrigarray[]** itself, remained unaffected, so we can use it all over again. Hence, on line 3 we pass **arOrigarray[..;..]** over to the other side of the expression (using slice syntax for WP8 compatibility). This is where **arOrigarray[]**'s actual data gets transferred into **arTemparray[]**. Then, on the other side, **arOrigarray[]** again appears, this time in the ***array*[0]** syntax and acting not as content but as form. Instead of furnishing data, **arOrigarray[]** now furnishes structural information, mere index ranges (ironically, for its competitor **arTemparray[]** rather than itself):

```
arTemparray[ .. arOrigarray[0 ; 1] ;  .. arOrigarray[0 ; 2] ] = arOrigarray[ .. ; .. ]
```

Adding Rows

OK, that's fine for augmenting the columns. What if you wanted to only increase the amount of *rows*, leaving columns the same? After you've done the initial multiD work, it's actually not a big adjustment. All you would need to change would be the placement of **+n** in the second line so that it adds rows but leaves the columns alone. Here's the new code for line 2...

```
    declare arTemparray[ arOrigarray[ 0 ; 1 ] + n ; arOrigarray[ 0 ; 2 ] ]
```

...and the function as a whole would look like this (sporting an appropriate new moniker):

```
Procedure ExtendRows2D (&arOrigarray[] ; n)

    declare arTemparray[ arOrigarray[0 ; 1] + n ; arOrigarray[0 ; 2] ]

    arTemparray[ .. arOrigarray[0 ; 1] ;  .. arOrigarray[0 ; 2] ] =
    arOrigarray[ .. ; .. ]

    arOrigarray[ ] := arTemparray[ ]
EndProc
```

Adding Both

Finally, to augment both rows and columns at the same time, we need give the function another parameter (a companion to **n**) so we can specify a new number of both rows and columns. It's a bit more complicated because this time the change affects line 1 as well as line 2, but line 3 remains the same. It's still not horrendously difficult to see how the macro is working. The code would look like this:

```
Procedure ExtendDim2D ( &arOrigarray[ ]; m; n )

    declare arTemparray[ arOrigarray[0 ; 1] + m; arOrigarray[0 ; 2] + n ]

    arTemparray[ .. arOrigarray[ 0 ; 1 ];  .. arOrigarray[ 0 ; 2 ] ] =
    arOrigarray[ .. ; .. ]
```

```
        arOrigarray[ ] := arTemparray[ ]
    EndProc
```

It's kind of a waste to have three functions running around, though. Luckily we can use the last version to handle all three contingencies. Remember, we've got two parameters now, both **m** and **n**. If you don't want to add rows, then simply pass **0** for **m** at function call; if you don't want to add columns, pass **0** for **n**; likewise, for whichever dimension you wish to augment, pass the desired number of elements, or pass numbers to each one, making both dimensions bigger simultaneously.

Here are some examples. Wanna increase columns by two? Then call **ExtendArray2D()** like this:

```
    ExtendArray2D ( &arOrigarray[ ] ;  0 ; 2)
```

How about adding 3 rows?

```
    ExtendArray2D ( &arOrigarray[ ] ; 3 ; 0)
```

And finally, here's how to increase rows by 1 and colums by 2:

```
    ExtendArray2D ( &arOrigarray[ ] ; 1 ; 2)
```

Alternatives Don't Work

You might think that other, simpler forms of syntax should work to extend an array dimension. Why multiple lines of code? Why not just add an empty value to the array like this...

```
    arOrigarray[ ] := arOrigarray[ ] + { "" }
```

...or like this?

```
    arOrigarray[ ] := { arOrigarray[ ] ; "" }
```

Neither works, however. The first expression does nothing whatsoever; it neither adds anything to the array, nor even causes an error message. In WP9 and above you can in fact use operators like **+**, **-**, **/**, or ***** to perform operations on an already existing array, but the operation applies to every single element in each dimension in the array, not to the array as an entire object. So **arOrigarray[]:=arOrigarray[] + {""}** fails to add a *new* element to the array. It just adds an empty string to whatever values the old elements already have. And if you're interested in learning other programming languages, forget these operators anyway; their capability to work with arrays in any manner at all is unique to PerfectScript.

The second expression at least generates the compile-time syntax error "Array indexes or dimensions expected." You can get rid of this error message by using a comprehensive slice in the

array on the right of the expression (the one inside the curly braces, e.g. if the original array had three elements, you could type **`arOrigarray[] := {arOrigarray[1..3]; ""}`**) and your code will compile. After that, however, you'll get a runtime error telling you that "All elements must be simple values." This occurs because the curly braces—**`{`** and **`}`**—signify that you're building an array with the direct-assignment method, using an array literal. This expression, then, tells the macro engine to put three *different* values (the three values contained in **`arOrigarray[1..3]`**) in the new array's first element. But a single array element can hold only a single ("simple") value, not a complex variable that holds multiple values, so you get the message and the macro quits. There may be other programming languages that automatically parse arrays into other arrays, or allow a single array element to hold an entire array (in other words, nested arrays), but PerfectScript certainly doesn't.

Flashback to the 50s: 3D!

It's an easy matter to change our 2D function to handle 3D array extensions—just use another parameter, such as "p", and add the extra dimension to the first two lines in the statement block (making sure to expand the second line's **`arOrigarray[]`** transfer into three slices for WP8 compatibility):

```
Procedure ExtendDim3D ( &arOrigarray[] ; n ; m ; p)

    declare arTemparray
    [
        arOrigarray[ 0 ; 1 ] + n;
        arOrigarray[ 0 ; 2 ] + m;
        arOrigarray[ 0 ; 3 ] + p
    ]

    arTemparray
    [
        .. arOrigarray[ 0 ; 1 ];
        .. arOrigarray[ 0 ; 2 ];
        .. arOrigarray[ 0 ; 3 ]
    ] = arOrigarray[ .. ; .. ; .. ]

    arOrigarray[ ] := arTemparray[ ]
EndProc
```

4D and Beyond

Well, this is fine so far. Unfortunately, to handle 4D arrays we'd have to add yet another parameter and yet another extra dimension to the first two lines in the statement block; to handle

5D arrays, we'd need to go through the same process again; and so on until we finally reached the maximum of ten dimensions. Not only that, but we'd have at least eight functions running around, one for the first three Ds and seven for the other six allowable dimensions. There's nothing that can be done about the additional material in the statement block, but wouldn't it be great if we could just have a single dimension-extender function that handled everything from a simple 1D polliwog to a complex 10D monster and did so without requiring an obscene number of params? "It would be nice," JDan mused, "if you didn't have to change the call format to handle larger-dimension arrays."

Doing this would be relatively easy if user-defined functions accepted repeating and optional parameters, but alas, such flexibility is limited to the native PerfectScript commands. JDan didn't let that stop him, however. Ratcheting up the creativity, he developed a function that fills the bill by reducing the parameters down to just a single pair: the array we want to extend (we could hardly jettison that), and then *another array* containing the extension values (i.e. dictating how much we want to extend the origarray). This parameter-array basically functions as a repeating parameter, faking out the macro system—JDan is the man! (The only real difference between the routine in this document and JDan's original is that I've made this one a procedure rather than a function.)

The New Call

When calling the new procedure, which we'll call **`ExtenDim()`**, you'll of course need a working, active origarray for the first param (otherwise you wouldn't be using the procedure!). The new second param is a pair of braces containing the values of a temparray. (You won't actually give the temparray a name in the function call, just its values. The temparray will be formally declared inside the function.) The temparray contains the amount of space we want to add to our "real" array. It is unidimensional, but the amount of elements in its single dimension can vary each time. The exact number will depend on the dimensionality of the origarray, the first parameter (the array we want to extend). If the array in question is two-dimensional, the second-parameter temparray will only have 2 elements in it, thus an index range of 2. But if we're working on an 8D array, we'll give the temparray 8 elements, thus an index range of 8. In other words, while the temparray itself is a normal unidimensional array, it requires the same number of *elements* as the first parameter has *dimensions*. This is important to remember, because if you're only planning to extend a single dimension of the origarray, you might be tempted to pass only a single element to the second parameter (the temparray). That won't work with this function. Let's say you have a 5D array of whose dimensions you only plan to expand the last, bottom-most (the columns) by 3 elements. You might be tempted to make the array "{3}". Bzzzt! All five dimensions must be represented by individual elements in the second parameter. In the case of such a 5D array, the 1D temparray's first 4 elements would be 0 (you only want to extend the last, fifth dimension), so the full array would be **`{0;0;0;0;3}`**. As another example, let's say we have a 5D array whose first three dimensions we want to extend but whose last two we want to keep the same. More specifically, in the first dimension we want two more elements, five in the second dimension, and four in the third (but—again—no change in the last two dimensions). Our second param would therefore work out to **`{2;5;4;0;0}`**. In this case, we'd be adding a bunch more branches to an array but not augmenting the size of the last two dimensions, the bottom-line rows and columns; basically, we're creating a bunch more folders and thus ultimately creating

more bottom-line "files," but we aren't making the number of files in *each "folder"* any bigger like we would if we had extended the last (fifth) dimension, the "columns." (In other words—to use the command-line file analogy described up in "Conceptualizing Arrays | Using Arrays"—if you were to augment the last dimension, you'd be changing the number of actual pieces of data, the "files," that are in those "folders" of the last dimension; otherwise, the number of "files" per "folder" stays the same (although of course the total number of files goes up precisely because there are more folders). Again, if any dimensions except the last are augmented, what changes is the number of "folders," not the number of "files" in each of them.)

And that's all there is to the new procedure call. Here's an example of a **`[4;3]`** lump-sum directly-assigned array...

```
arOrigarray[ ]={ { "A"; "B"; "C" } ; { "D"; "E"; "F" } ; { "G"; "H"; "I"} ; { "J"; "K"; "L" } }
```

...which will now be successively augmented by two columns, one row, and finally another row and two more columns:

```
//  Increase columns by 2:
ExtenDim ( &arOrigarray[ ]; {0; 2} )

//  Increase rows by 1:
ExtenDim ( &arOrigarray[ ]; {1; 0} )

//  Increase rows by 1 and colums by 2:
ExtenDim ( &arOrigarray[ ]; {1; 2} )
```

The New Procedure

And now for the new procedure itself. First, we state the procedure name and parameters. The first param is our old friend ***`&origarray`***, but as we know, the second will now also be an array (named **`e`** for "extension")...

```
Procedure ExtenDim ( &arOrigarray[ ] ; e[ ] )
```

...the array that gets the direct literal assignment (what I like to call "lump-sum" assignment) discussed in the previous subsection ("The New Call").

Before we do any real work, we should—as always—do some error-checking. Remember, the second param has to have the same number of elements as the origarray has dimensions. To ensure that some dum...er, inattentive macro user doesn't mess this up, we'll deploy the special array operator ***`array[0;IndexOption]`*** in an **`If`** statement. You'll recall that the **`-1`** enumerator for the array operator's second param reports the number of an array's dimensions, and of course the bare ***`array[0]`*** syntax, without that second param, will give you an array's total elements. So we just...

1. reference the new 1D temparray and count its elements with ***array*[0]**;

2. reference the origarray and count its dimensions with **-1**;

3. then use the negative operator **!=** to see whether they match...

```
If ( arOrigarray[ 0 ; -1 ] !=  e[ 0 ] )
```

4. ...and if they don't, the **If** fires its emergency innards:

```
Prompt ("You Didn't Enter An Appropriate Number")
Pause
Quit // quit the procedure
Endif
```

If everything checks out OK, though, we can start getting our hands dirty and actually do the extending. We'll reuse the **-1** enum of the special array operator, putting it inside a **Switch-CaseOf** series both to tell PerfectScript what the origarray's dimensions are and then to extend it appropriately.

First, we pass the origarray's dimension count into the **Switch** with the **-1** enum of the ***array*[0;*IndexOption*]** operator...

```
Switch ( arOrigarray[ 0 ; -1 ] )
```

...which makes **Switch** test for the number of dimensions that the origarray has. The Switch will thus always return a number from 1 to 10. For their part, the **CaseOf** "answers" will be numeric: **CaseOf 1**, **CaseOf 2**, **CaseOf 3**, and so forth. Thus, if the origarray has one dimension, **Switch** will return a 1, and **CaseOf 1** will fire; if two dimensions, **Switch** returns a 2, and **CaseOf 2** fires; and so on. We'll create ten **CaseOf**s, one for each of the ten allowable dimensions, so no matter how big the origarray, this procedure's got it covered.

Each **CaseOf**'s statement block will comprise two lines, our now-familiar declaration and structuring of the temparray and then the population of the temparray with the origarray's data. The first line, where we declare and structure the temparray (without populating it) is a bit different now because instead of adding a number passed directly into a normal var, we have the **e[]** paramarray, but otherwise it works the same as before.

For example, let's say that we have a unidimensional array named "arGoober." During the course of the macro it's in, we'll need to make the array four elements bigger. Given this scenario, we'd call the procedure like this: **arGoober[] = ExtenDim (&arGoober[];{4})**. Because our origarray is unidimensional, we give **e[]** only a single value ("4") in the call, so for all practical purposes **e[]**is no different than a normal var—it does exactly the same thing as **n** did, way back in our very first, most basic function **Extend()** (see above, **Extending Unidimensional Arrays | Becoming Functional**).

So here we are inside the function. Just as Mudville had Casey, **ExtenDim()** has **CaseOf**. When the **Switch** test determines that we have an array with only a single dimension, **CaseOf 1** is called up to bat:

```
CaseOf 1:
    declare arTemparray[ arOrigarray[ 0 ; 1 ] + e [ 1] ]
```

First, it carefully assesses the pitcher, noticing that the windup indicates one of those array curveballs: a new array, **arTemparray[]**, is getting declared. To meet the challenge, **CaseOf 1** hefts an ***array[0;IndexOption]*** Louisville Slugger, grabbing the number of elements in the origarray's first dimension in the form **arOrigarray[0;1]**. With a few warmup swings, he steps up to the plate. But as the pitch comes screaming in, **CaseOf 1** sizes it up and decides to use a special bat—not just any old **arOrigarray[0;1]**, but rather a special one with **e[1]** cork inside.

Some of the fans in the stadium might think this choice redundant. Sure, **e[1]** references the first element in **e[]**, but there's only a single element in **e[]** anyway. The entire index range of **e[]** is 1, so aren't **e[1]** and **e[]** the same, logically speaking? Couldn't you just save a character and use **e[]**? Well, in the grand logical scheme of things maybe **e[1]** and **e[]** are the same, but no, **CaseOf 1** can't use **e[]** here. Remember, single value or not, we're dealing with an array. The value contained therein can only be accessed via its index tag—i.e. by using the ***array[index]*** syntax, in this case **e[1]**. If **CaseOf 1** tried to get by with **e[]**, he'd strike out. So he flexes his binaries and swings with **e[1]**, which tags the array's single element, which in turn contains a value of **4**—so the origarray size gets augmented by four and the resulting sum is passed into **arTemparray[]**. Whack! It's a fast grounder to left field!

While this gets **CaseOf 1** some applause as he slides in to first base, he's still along way from home. The temparray has been declared and given the size mandated in the procedure call, but to score, **CaseOf 1** now needs a temparray populated with the origarray's actual data. Here's the code for achieving this...

```
arTemparray[ .. arOrigarray[ 0 ; 1 ] ] = arOrigarray[ .. ]
```

...and here are the behind-the-scenes details:

1. Using slice syntax for WP8 compatibility, **CaseOf 1** grabs the entire origarray (**arOrigarray[..]**) and copies it to memory, stealing his way to second.

2. Then he moves over to the other side of the equals sign, giving **arTemparray[]** a sliced index equal to the size of the origarray's first dimension (**..arOrigarray[0;1]**, which of course is actually the entire origarray because if this CaseOf fired, it only *has* a single dimension). Suddenly **CaseOf**'s on third!

3. Next he passes **arOrigarray[..]** into **arTemparray[..arOrigarray[0;1]]**, so now the temparray **arTemparray[]**

has not only the extra number of elements we asked for but also the origarray's data (down toward the bottom). **arTemparray[]** is ready to be passed back out to the larger macro and get its new data! **CaseOf** scores!

The other **CaseOf**s seem more complicated, but they're really just more of the same. Here's **CaseOf 2**:

```
CaseOf 2:
    declare arTemparray[ arOrigarray[ 0 ; 1 ] + e[ 1 ] ; arOrigarray[ 0 ; 2 ] +
    e [ 2 ] ]
    arTemparray[ .. arOrigarray[ 0 ; 1 ] ; .. arOrigarray[ 0 ; 2 ] ] =
    arOrigarray[ .. ; .. ]
```

For array newbies, perhaps the most potential confusion lies in the first line where **arTemparray[]** is declared with **arOrigarray[0;1] + e[1]; arOrigarray[0;2] + e[2]**. After all, when you call the procedure, you're probably not passing in "1" for **e[]**'s rows nor "2" for its columns, but rather something like "{4;3}"—i.e. four rows and three columns (or, if you want to use the command-line file analogy, four folders and three files inside each of them). So why is the first line of the **CaseOf** hard-coded with **e[1]** and **e[2]**? Shouldn't it be **e[4]** and **e[3]**? Well, no. *Structuring* an array is a different operation than *referencing* an array. The direct-assignment "{4;3}" sets up an array that has one dimension containing the two "files" 4 and 3. You've just structured the array. But from that point on, you can reference it to use the data in the array (the 4 and/or the 3), and when you do that, you usually don't want to use all the data at once. You don't want the 4 and the 3 together; you want one or the other, not both. The programming gurus set up array references to accommodate this general fact of usage, so when you reference an array, the number you type inside the brackets refers to one particular element, not the dimension as a whole; in addition, that number *is not the data*. It's the index, and you'll recall that an index number is a label, a tag, not the actual contents. **e[1]** is the index of 4, the first value in **e[]**, while **e[2]** is the index of 3, the second value. So we don't refer to the values directly—we access them via their indices. And that's a good thing. Sure, if we typed "{4;3}" for our second parameter at function call, we want to add four rows and three columns, but remember that we may want to change that in the future; next time, we may want to add 1 row and 25 columns, or 5 rows and 2 columns, or whatever. In order to have a procedure flexible enough to add however many rows or columns we want at any given time, we need to soft-code everything inside the function that we might want to vary on different occasions. The **[1]**,**[2]**,**[3]** (etc.) syntax for indices, in which a given dimension's values are indexed in ascending order, allows us to do that (and indeed gives us no other choice).

OK. Look back to the first **CaseOf** for a moment, then note how, here in the second **CaseOf**, we simply keep reeling out the augmentation operations in the first line. Now that we have two dimensions, the new dimension (rows) is the first and the columns are now the second, and the numbers change to reflect this status, but the augmentation operation itself remains the same and will do so all the way up to the tenth dimension under **CaseOf 10**. Likewise with the second line's array-slice populating operations. The key, really, lies more in the procedure call than in the lockstep **CaseOf**s. A quick reminder might be in order here: to adduce one of the sample

procedure calls already described above (in **The New Call**), if we had a 2D origarray named **arSchmuck[]** and we wanted to increase the last (most fundamental) dimension by two, we'd type **arSchmuck[] = ExtenDim (&arOrigarray[]; {0;2})**, giving **ExtenDim()** a 2D array, **{0;2}** (not a unidimensional one), for its second param, *but making the first dimension have no elements*. The last one has two. Because we did it this way, down in the **CaseOf** the first restructuring operation (in the first line) won't do any restructuring; the expression is literally working with nothing, 0. It goes through the calculations but doesn't augment the rows; **arTemparray[]** ends up with the exact same number of rows as the origarray. Essentially, its row augmentation operation is ***OrigarrayRows* + 0**. Not so with columns, however. Because we specified the second (last) element of the parameter as **2**, the second augmentation operation is essentially ***OrigarrayColumns* + 2**; columns get extended by 2.

And so it goes, from **CaseOf 3** to **CaseOf 10**. Finally, right before the **Switch** finishes up, we use the **Switch**'s optional **Default** feature to do a last bit of error-checking. **Default** fires if none of the **CaseOf**s match the **Switch** test. There's no legitimate reason for that to happen with **ExtenDim()** because we've got all ten dimensions covered; if it does happen, it'll be because bonehead coders messed up the first parameter somehow. And since bonehead coders do exist in this imperfect world (I am frequently one of them), we might as well use **Default** to allow for them:

```
                Default:
                        Prompt ("You passed a faulty value into procedure
                                ExtenDim()'s first parameter. Please check your
                                work.")
                        Pause
                        Quit // Shut the macro down so the bozo can fix the code
        Endswitch
```

And that's pretty much it. Just top things off by doing the usual overwrite...

```
        arOrigarray[ ] := arTemparray[ ]
EndProc
```

...and we're done.

The Big Kahuna

Here's the whole thing, accompanied by several test messages to demonstrate how it works (suitable for cutting-and-pasting into WP and compiling as a macro). I've included two versions here, identical except for the technique of passage by reference which is used by the first snippet, a procedure, and not by the second, a function:

```
// THE PROCEDURE WITH PASSAGE BY REFERENCE
Procedure ExtenDim (&arOrigarray[]; e[])
if (arOrigarray[0;-1] != e[0])
        Prompt ("Mismatch between original array dimensions and parameter array elements;
        macro will now terminate.")
        Pause
        Quit
endif

switch (arOrigarray[0;-1])

        caseof 1:
                declare arTemparray[arOrigarray[0;1]+e[1]]
                arTemparray[..arOrigarray[0;1]] = arOrigarray[..]

        caseof 2:
                declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2]]
                arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]] = arOrigarray[..;..]

        caseof 3:
                declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
                arOrigarray[0;3]+e[3]]
                arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3]] =
                arOrigarray[..;..;..]

        caseof 4:
                declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
                arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]]
                arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
                ..arOrigarray[0;4]] = arOrigarray[..;..;..;..]

        caseof 5 :
                declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
                arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]; arOrigarray[0;5]+e[5]]
                arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
                ..arOrigarray[0;4]; ..arOrigarray[0;5]] = arOrigarray[..;..;..;..;..]
```

```
caseof 6 :
        declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
        arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]; arOrigarray[0;5]+e[5];
        arOrigarray[0;6]+e[6]]
        arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
        ..arOrigarray[0;4]; ..arOrigarray[0;5]; ..arOrigarray[0;6]] =
        arOrigarray[..;..;..;..;..;..]

caseof 7:
        declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
        arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]; arOrigarray[0;5]+e[5];
        arOrigarray[0;6]+e[6]; arOrigarray[0;7]+e[7]]
        arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
        ..arOrigarray[0;4]; ..arOrigarray[0;5]; ..arOrigarray[0;6]; ..arOrigarray[0;7]] =
        arOrigarray[..;..;..;..;..;..;..]

caseof 8:
        declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
        arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]; arOrigarray[0;5]+e[5];
        arOrigarray[0;6]+e[6]; arOrigarray[0;7]+e[7]; arOrigarray[0;8]+e[8]]
        arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
        ..arOrigarray[0;4]; ..arOrigarray[0;5]; ..arOrigarray[0;6]; ..arOrigarray[0;7];
        ..arOrigarray[0;8]] = arOrigarray[..;..;..;..;..;..;..;..]

caseof 9:
        declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
        arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]; arOrigarray[0;5]+e[5];
        arOrigarray[0;6]+e[6]; arOrigarray[0;7]+e[7]; arOrigarray[0;8]+e[8];
        arOrigarray[0;9]+e[9]]
        arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
        ..arOrigarray[0;4]; ..arOrigarray[0;5]; ..arOrigarray[0;6]; ..arOrigarray[0;7];
        ..arOrigarray[0;8]; ..arOrigarray[0;9]] = arOrigarray[..;..;..;..;..;..;..;..;..]

caseof 10:
        declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
        arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]; arOrigarray[0;5]+e[5];
        arOrigarray[0;6]+e[6]; arOrigarray[0;7]+e[7]; arOrigarray[0;8]+e[8];
        arOrigarray[0;9]+e[9]; arOrigarray[0;10]+e[10]]
        arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
        ..arOrigarray[0;4]; ..arOrigarray[0;5]; ..arOrigarray[0;6]; ..arOrigarray[0;7];
        ..arOrigarray[0;8]; ..arOrigarray[0;9]; ..arOrigarray[0;10]] =
        arOrigarray[..;..;..;..;..;..;..;..;..;..]

        default:
                Prompt ("Something is wrong. The macro will now terminate.")
                Pause
```

```
                    Quit
        endswitch
        arOrigarray[] := arTemparray[]
EndProc

//A [2;2;3] array:

arTest[ ] =
{
        {{ "John"  ;  "M"  ; 44 } ;
        { "Mike"   ;  "S"  ; 32 }} ;
        {{ "Frank" ;  "S"  ; 19 } ;
        { "Ron"    ;  "M" ; 35 }}
}

MessageBox (; "Unextended Array"; "Array arTest contains " + arTest[0;-1] + " dimension(s) and
a total of " + arTest[0] + " elements. The elements in the bottom-most dimension, last folder, are
" + arTest[2;2;1] + ", " + arTest[2;2;2]  +", and " + arTest[2;2;3]  + ".")

ExtenDim (&arTest[]; {0;0;1})
arTest[2;2;4] = "Y"

MessageBox (; "Newly Extended Array"; "Array arTest has been modified; it still contains " +
arTest[0;-1] + " dimensions, but now boasts a total of " + arTest[0] + " elements. The latest new
element is " + arTest[2;2;4] + ".")

// A [2;2;3;4] array:

declare arTest[] =
{
        {
                {
                        { 111;112;113;114} ; {121;122;123;124} ; {131;132;133;134}
                } ;
                {
                        { 211;212;213;214} ; {221;222;223;224} ; {231;232;233;234}
                }
        };
        {
                {
                        {"A";"B";"C";"D"} ; {"E";"F";"G";"H"} ; {"I";"J";"K";"L"}
                } ;
                {
                        { "M";"N";"O";"P"} ; {"Q";"R";"S";"T"} ; {"U";"V";"W";"X"}
                }
        }
```

```
}

MessageBox (; "Unextended Array"; "Array arTest contains " + arTest[0;-1] + " dimensions and a total of " + arTest[0] + " elements. The elements in the bottom-most dimension, last (fourth) folder, are " + arTest[2;2;3;1] + ", " + arTest[2;2;3;2]  + ", " + arTest[2;2;3;3] +", and " + arTest[2;2;3;4]  + ".")

ExtenDim (&arTest[]; {0;0;0;1})
arTest[2;2;3;5] = "Y"

MessageBox (; "Newly Extended Array"; "Array arTest has been modified; it still contains " + arTest[0;-1] + " dimensions, but now boasts a total of " + arTest[0] + " elements. The latest new element is " + arTest[2;2;3;5] + ".")

// A [1;1;1;1;1;1;1;4;2;2] array:

arTest[] =
{{{{{{{
                {
                        {{"A";"B"};{"a";"b"}};
                        {{"C";"D"};{"c";"d"}};
                        {{"E";"F"};{"e";"f"}};
                        {{"G";"H"};{"g";"h"}}
                }
}}}}}}}

MessageBox (; "Unextended Array"; "array arTest contains " + arTest[0;-1] + " dimensions and a total of " + arTest[0] + " elements. The values (files) from that part of the tenth dimension which is contained in the fourth folder of the eighth dimension (which contains both folders of the ninth dimension) are " + arTest[1;1;1;1;1;1;1;4;1;1] + ", " + arTest[1;1;1;1;1;1;1;4;1;2]  + ", " + arTest[1;1;1;1;1;1;1;4;2;1] +", and " + arTest[1;1;1;1;1;1;1;4;2;2]  + ".")

ExtenDim (&arTest[]; {0;0;0;0;0;0;0;0;0;1})
arTest[1;1;1;1;1;1;1;4;2;3] = "I"

MessageBox (; "Newly Extended Array"; "array arTest still contains " + arTest[0;-1] + " dimensions but now boasts an increased total of " + arTest[0] + " elements. The newest element is " + arTest[1;1;1;1;1;1;1;4;2;3] + ".")

// THE FUNCTION, WITH NORMAL VARIABLE BEHAVIOR:

Function ExtenDim (arOrigarray[]; e[])
        if (arOrigarray[0;-1] != e[0])
                Prompt ("Mismatch between array geometry, and expansion size")
                pause
                return ({0})
```

```
        endif

        fornext (i; 1; e[0])
                if (arOrigarray[0;i]+e[i] > 32767)
                        Prompt ("Index " + i + " will exceed limit of 32767")
                        pause
                        return ({0})
                endif
endfor

        switch (arOrigarray[0;-1]) // or switch (e[0])

        caseof 1:
                declare arTemparray[arOrigarray[0;1]+e[1]]
                arTemparray[..arOrigarray[0;1]] = arOrigarray[..]

        caseof 2:
                declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2]]
                arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]] = arOrigarray[..;..]

        caseof 3:
                declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
                arOrigarray[0;3]+e[3]]
                arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3]] =
                arOrigarray[..;..;..]

        caseof 4:
                declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
                arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]]
                arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
                ..arOrigarray[0;4]] = arOrigarray[..;..;..;..]

        caseof 5 :
                declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
                arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]; arOrigarray[0;5]+e[5]]
                arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
                ..arOrigarray[0;4]; ..arOrigarray[0;5]] = arOrigarray[..;..;..;..;..]

        caseof 6 :
                declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
                arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]; arOrigarray[0;5]+e[5];
                arOrigarray[0;6]+e[6]]
                arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
                ..arOrigarray[0;4]; ..arOrigarray[0;5]; ..arOrigarray[0;6]] =
                arOrigarray[..;..;..;..;..;..]
```

```
        caseof 7:
                declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
                arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]; arOrigarray[0;5]+e[5];
                arOrigarray[0;6]+e[6]; arOrigarray[0;7]+e[7]]
                arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
                ..arOrigarray[0;4]; ..arOrigarray[0;5]; ..arOrigarray[0;6]; ..arOrigarray[0;7]] =
                arOrigarray[..;..;..;..;..;..;..]

        caseof 8:
                declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
                arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]; arOrigarray[0;5]+e[5];
                arOrigarray[0;6]+e[6]; arOrigarray[0;7]+e[7]; arOrigarray[0;8]+e[8]]
                arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
                ..arOrigarray[0;4]; ..arOrigarray[0;5]; ..arOrigarray[0;6]; ..arOrigarray[0;7];
                ..arOrigarray[0;8]] = arOrigarray[..;..;..;..;..;..;..]

        caseof 9:
                declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
                arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]; arOrigarray[0;5]+e[5];
                arOrigarray[0;6]+e[6]; arOrigarray[0;7]+e[7]; arOrigarray[0;8]+e[8];
                arOrigarray[0;9]+e[9]]
                arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
                ..arOrigarray[0;4]; ..arOrigarray[0;5]; ..arOrigarray[0;6]; ..arOrigarray[0;7];
                ..arOrigarray[0;8]; ..arOrigarray[0;9]] = arOrigarray[..;..;..;..;..;..;..;..;..]

        caseof 10:
                declare arTemparray[arOrigarray[0;1]+e[1]; arOrigarray[0;2]+e[2];
                arOrigarray[0;3]+e[3]; arOrigarray[0;4]+e[4]; arOrigarray[0;5]+e[5];
                arOrigarray[0;6]+e[6]; arOrigarray[0;7]+e[7]; arOrigarray[0;8]+e[8];
                arOrigarray[0;9]+e[9]; arOrigarray[0;10]+e[10]]
                arTemparray[..arOrigarray[0;1]; ..arOrigarray[0;2]; ..arOrigarray[0;3];
                ..arOrigarray[0;4]; ..arOrigarray[0;5]; ..arOrigarray[0;6]; ..arOrigarray[0;7];
                ..arOrigarray[0;8]; ..arOrigarray[0;9]; ..arOrigarray[0;10]] =
                arOrigarray[..;..;..;..;..;..;..;..;..;..]
        EndSwitch

        arOrigarray[] := arTemparray[]
        Return (arOrigarray[])
EndFunc
```

This second version, a nonreferential function instead of a referential procedure, is important for two reasons: first, a function rather than a procedure is necessary for use with the next step (adding values to the array inside the routine instead of out in the main macro). Second, the next step requires that WP8 users employ a nonreferential function. On the WordPerfect Universe forum JDan created the function with passage by reference to reduce memory overhead, but my copy of WP8 (SP7) generated the standard array error, "There is not enough memory available to

play the macro. Close down some other applications and try to play the macro again." JDan thought he had solved this problem by introducing the use of consistent array slices (see above, "Dimensionality Demasked | Slices | WP8 Bug"), but I'm using his improved, solution-based function, and when it includes the passage by reference, it still generates the error because of the one-dimensional collapse problem (see **Dimensionality Demasked | Slices | WP8 Bug**, above). So WP8 users will have to forego passing by reference.

Value-ing Your Extensions

It's great that we've got a function for adding elements to any pre-existing dimension of any array. But when the dust settles and the function's finished, the origarray is still essentially its old self; sure, it's got more space, but that space isn't occupied. To give values to the new elements, we've been doing hard-coded direct assignments—and even neophyte programmers know that such a procedure limits flexibility within this macro itself and portability to other macros. Not only that, it would save coding time if the function could take care of inserting the new values along with extending the array dimensions. And JDan came through one more time with the answer. It's a new function, but don't throw up your hands and wonder why we just spent so much time on **`ExtenDim()`**—never fear, we'll still be using it.

JDan took macro-writers on the WordPerfect Universe forum through the development of a new value-adding function step by step, explaining clearly what steps were involved and how the code was working. This process may seem tedious to coders who just want a "widget" they can paste into their macro and use without further ado, but you can't effectively use this widget unless you understand how it works because you won't know how to call it correctly. And if you're writing complex macros that juggle lots of complex data, JDan's widget—a function dubbed "AddElements()"—just might become indispensable once you've mastered it.

Inflating the Flat

So let's begin with a basic function that adds values to just a flat, normal, one-dimensional array.

1. Here are the function name and parameters:

```
FUNCTION AddElements1D ( arOrigarray[ ] ; arExtra[ ] )
```

The function's first param is self-explanatory, but the second, called **`arExtra[]`**,is new; this time it's an array containing not the number of elements you want to add to a dimension, but instead the *actual values* you wish to add to the origarray.

2. The second line features a familiar friend:

```
arNew[ ] = ExtenDim ( arOrigarray[ ] ; { arExtra[ 0 ] } )
```

The cool thing about passing the values themselves into the 2nd parameter is that an array with those values will, by definition, have the exact number of elements by which you must extend the origarray's dimension so it can accept the additional values. As a result, you can use **arExtra[]** not just to hold the values you want inserted into the origarray, but you can also use it as the second parameter in **ExtenDim()** for extending the origarray's dimensions—which, after all, must be done before new values can be added. So the first thing the function does is nest an **ExtenDim()** call, using **arExtra[0]** as the second param. This gives us a restructured array ready to receive new elements. Then we pass this into a new array, called **arNew[]**, which will quite literally be our new array—not merely a restructured array, but one that will eventually hold all those new values, our ultimate goal.

3. Then we take the array that holds the additional values (the parameter **arExtra[]**), make it into a (total) slice, and pass it into a top slice of array **arNew[]**:

```
arNew[ arOrigarray[ 0 ] +1 .. ] := arExtra[ .. ]
```

This "top slice" in **arNew[]** begins at the end of the origarray's number of elements and runs through to the end of the array, which of course comprises a range precisely equal to that of **arExtra[]**. We keep the top of the **arNew[]** slice unspecified so that it'll automatically default to the absolute upper limit of the array's index range; the bottom of the **arNew[]** slice, however, will be **arOrigarray[0]+1**, so the entire slice will be **[arOrigarray[0]+1 ..]**.

If, like me, you happen to be mathematically challenged, the reason for this might elude you at first. The **arOrigarray[0]** is easy enough to understand—the total number of array elements—but, you might ask, isn't it enough? Why add 1 on top of that? Well, let's say your origarray is **arTest[]={"A";"B";"C";"D";"E"}**. In this case, **arTest[0]** is 5, so if you typed "arOrigarray[0].." for your slice, "5.. " is what you would actually be specifying. And you don't want that; the element arTest[5] already has a value (the letter E), and the purpose of the function isn't to overwrite already-existing elements but rather to add brand-new elements and put brand-new values into *them*, keeping all the old stuff in the bargain. So slicing **arTest[]** with "arOrigarray[0].." is syntactically OK—it's a valid slice—but it's a semantic error in the context of this particular macro. The macro won't generate a compile-time error message, but you'll have written a programming error into the code. To avoid this, insert that "+1" and everything will be fine; instead of beginning your slice at the last old element, you'll start it at the first new element you added, which is exactly what you want. And when the line has done its job, **arNew[]** has values for all the extra space it got back on line 2.

4. Now we return our new array back out to the main macro...

```
RETURN( arNew[ ] )
```

...and the finished function looks like this:

```
FUNCTION AddElements1D ( arOrigarray[ ] ; arExtra[ ] )

    arNew[ ] = ExtenDim ( &arOrigarray[ ] ; { arExtra[ 0 ] } )

    arNew[ arOrigarray[ 0 ] +1 .. ] := arExtra[ .. ]

    RETURN( arNew[ ] )

ENDFUNC
```

5. Finally, out in the main macro, we use the function call to overwrite the origarray with the return value of the function:

```
arTest[ ] = { 1 ; 2 ; 3 ; 4 ; 5 } //First, declare an array with 1 dimension and 5 elements; mebbe use it a bit; then...

arTest[ ] = AddElements1D ( arTest[ ] ; { 6 ; 7 ; 8 } ) // ...when the old array doesn't cut it anymore and you need those new values, extend the origarray and add the values (putting the value-array in the second parameter).
```

Now you have an array that is *both* 8 elements long instead of 5, *and* contains values in all 8 elements.

Revaluing 2Ds

Now 2 dimensions. Remember when we were developing **`ExtendColumns2D()`** and **`ExtendRows2D()`**, we asked ourselves whether we would be expanding the columns or the rows? Same deal here. Will we just be adding new columns of data, or new rows?

Adding Values to the Second Dimension (Columns)

Let's do columns first. Here's the array we want to change:

```
arTest[ ] = { {"A" ; "B" ; "C" ; "D"} ; {"E" ; "F" ; "G" ; "H"} }
```

Of course we make our origarray first. Here, as before, we use the lump-sum method, imparting structure above the last dimension with braces and the last dimension's structure (the bottom dimension) with the actual element values. We don't have to make the array in this manner, of course, but for instructional purposes it's the best method because it lets us see the entire array at a glance. The end result is a **`[2;4]`** origarray, with the two sets of 4 (the two columns) containing, respectively, the bottom-line, buck-stops-here values A through D and E through H.

Out in the main macro we make the function call, overwriting the origarray's data with the new structure and values:

```
arTest[ ] = AddElements2Dcolumns
            (
                    arTest[ ] ;  { {"I" ; "J" ; "K"} ; {"L" ; "M" ; "N"} }
            )
```

This isn't much like the first function's call, that's for sure. But we're still doing the same thing: the second parameter is still a lump-sum direct-assignment array, except now it's 2D instead of 1D. That's the only difference, and it has to be 2D because now we're trying to manipulate a 2D array. Keep in mind that this function will be designed only to change the second (columnar) dimension, not the first (the rows). And we need to specify the same number of values *per row*. We may only want to add one value, or we may want to add a hundred, but however many we decide on, we must put that many columnar values into *each* set of row braces.

What does that mean? It doesn't mean that we have to use the same number of columnar values that are in the origarray; it just means that we have to have a consistent number of columnar values within the paramarray itself. Let's say, for example, that we have a **`[3;2]`** array—**`{{"A";"B"};{"C";"D"};{"E";"F"}}`**—and we want to extend the columns by five. In that case, we need five values for each of those three "sets" of rows. Or to use the command-line file analogy, the first dimension (rows) has three folders. These folders contain the second dimension (columns): two files apiece. Remember, the second dimension in this array has an index range of 2 because it has two values, so each first-dimension "folder" contains two "files" (or each row spans two columns). In this function we can mess with the 2, the index range of the second dimension (the files or columns), but not with the 3, the index range of the first dimension (the folders or rows). So if we want to give 5 more values to the columns, we have to specify them for *all three rows*, e.g.:

```
{ {"G";"H";"I";"J";"K"} ; {"L";"M";"N";"O";"P"} ; {"Q";"R";"S";"T";"U"} }
```

In other words, we can't simply decide that we wanted to enlarge one of the columnar sets and pass in **`{{"G";"H";"I";"J";"K"}}`** alone as the second parameter, not worrying about the other two sets at this particular point in the (imaginary) macro; there's no way to specify which of the three sets we're targeting. It could be either the first, second, or third rows (or, in terms of the command-line file analogy, the first, second, or third folders), and since PerfectScript isn't intelligent enough to choose between them, it'll return an error. We could, however, pass empty strings to the other sets, e.g.:

```
{ {"G" ; "H" ; "I" ; "J" ; "K"} ; {"" ; "" ; "" ; "" ; ""} ; { "" ; "" ; "" ; "" ; ""} }
```

—and achieve the same goal. What's important is the placeholding.

OK, so once we've called the function and given it proper parameters, how does the function itself work? Actually, although much of it looks a lot different, the same principles that applied in the first, 1D function are also at work in each line of this one:

1. The function's first line remains exactly the same (aside from the new name for the function):

```
FUNCTION AddElements2Dcolumns ( arOrigarray[ ] ; arExtra[ ] )
```

There is a difference between the previous function and this one—what we pass to the second parameter at function call (a 2D array with the same number of rows as the origarray and the elements we want to add in the columns). But we've just covered that issue.

2. Remember, in the second line we only restructure the array rather than putting the new values in. Still, it starts getting sticky here, because now we use **`ExtenDim()`**, our old friend. We'll need to worry about our origarray's dimensionality not only in **`AddColumnElements2d()`**'s function call (in the main macro body), but also inside the function when calling **`ExtenDim()`**. Since **`AddColumnElements2d()`** is specifically for restructuring/adding values to a 2D array's second (columnar) dimension, we have to first make sure that the origarray is indeed 2D, then pass **`ExtenDim()`** a second parameter that will give us the appropriate restructuring. Given **`AddColumnElements2d()`**'s narrow, columns-only purpose, this param will always be **`{0; arExtra[0;2]}`**, with the first index ("0") acting as the origarray's first-dimension placeholder and the second index (using the special reserved array operator ***`array[0;IndexOption]`***) containing the number of elements we want to add to the origarray's second dimension (because **`arExtra[]`** is the paramarray, and at function call we passed it an array with the columnar values, so its columns of necessity contain the right number of elements).

Aside from that, this second line is much the same as it was in the previous function—pass the newly restructured array over to **`arNew[]`**...

```
arNew[ ] := ExtenDim ( &arOrigarray[ ] ; { 0 ; arExtra[ 0 ; 2 ] } )
```

3. ...which now, in the third line, receives values for the new portions of its structure. This line still works on the exact same principles as it did in the first function: we take the array that holds the additional values (the parameter **`arExtra[]`**), make it into a (total) slice, and pass it into a top slice of array **`arNew[]`** (a slice beginning with the origarray's column-total-number-of-elements-plus-one and running through to the end of the array, which of course comprises all the new spaces that just got added in line 2). So in line 2, **`ExtenDim()`** gave **`arNew[]`** the origarray's elements and the extra space; here in line 3, **`arExtra[]`** gives **`arNew[]`** values for that extra space:

```
arNew[ .. ; arOrigarray[ 0 ; 2 ] + 1 .. ] := arExtra[ .. ; .. ]
```

There is a difference, however. Because we're dealing with a 2D array, we must have two slices instead of one, and for that very reason we can't use the shortcut ***`array[0]`*** syntax of the reserved array operator to demarcate the lower limit of the second slice; we have to use the full-blown ***`array [0;X]`*** syntax—and of course, for WP8 compatibility, we must slice **`arExtra[]`** as many times as we have dimensions. Here's where we hit a puzzling snag for newbies (it was a snag for me, anyhow): since this is a column-extending function, the first slice (the row slice) should just be a placeholder. We want the rows to stay the same; we don't want to

do anything to them at all. There's a problem with this goal, though. There's no way to pass **arExtra[]**'s columns to **arNew[]**'s top column slice without touching **arNew[]**'s rows in the process. It just can't be done; PerfectScript requires an exact dimensional fit when passing data into arrays—we must pass data from *both* of **arExtra[]**'s dimensions into *both* of **arNew[]**'s. We simply cannot avoid writing into **arNew[]**'s rows. The good news is that it doesn't matter! Remember, *there's no data there to begin with anyway*. The row elements are just "folders"–the actual data, the "files," are in the column elements. Row elements just *hold* the "files." So we'll end up completely overwriting the dimension that we'd prefer not to mess with, but it's OK, because the only thing there anyway is the structural space denoted by the index range. So all we have to do to keep the same row structure is give **arExtra[]** the origarray's index range, and we did that at function call! So everything's cool. **arExtra[]**'s first dimension already exactly matches that of the origarray. (See the first part of this section, above.) And since that's the case, we can overwrite the origarray's rows with **arExtra[]**'s and not miss a beat, precisely because they're *exactly the same*.

4. The fourth line remains exactly the same as it was in **AddElements1D()**, and will remain the same into perpetuity, so we won't mention it again. Here's the function in its entirety, always remembering that the **arExtra[]** parameter now requires, at function call, a two-dimensional array in which the first dimension (rows, or "folders") are exactly the same as the origarray's, and the second dimension (columns, or "files") holds the elements we wish to add:

```
FUNCTION AddElements2Dcolumns ( arOrigarray[ ] ; arExtra[ ] )

    arNew[ ] := ReDimN ( &arOrigarray[ ] ; { 0 ; arExtra[ 0 ; 2 ] } )

    arNew[ .. ; arOrigarray[ 0 ; 2 ] + 1 .. ] := arExtra[ .. ; .. ]

    RETURN( arNew[ ] )
ENDFUNC
```

...and when **AddElements2Dcolumns()** is finished doing its thing, **arTest[]** will now be **{{"A";"B";"C";"D";"I";"J";"K"};{"E";"F";"G";"H";"L";"M";"N"}}**.

Adding Values to the First Dimension (Rows)

Let's call the function for the rows **AddElements2Drows()**. Its call will seem a bit odd because *we're still adding elements to the columns*, even though the function's "official" job is to expand the rows. What happens is that the row index range, the index range of the first dimension, is increased, but the actual values go to the columns—the second dimension—and must be accessed as such. The index range of the columnar dimension remains the same, but because there are now more rows, there are more values in the columns. If you're getting confused, don't worry. This is where the command-line file analogy makes a great deal more sense than the table analogy, at least to me, so let's switch to the former. (If you need to discuss this function with someone else who's never heard of the command-line file analogy, you can always translate into the row-column terminology after you yourself understand what's going on.)

First off, recall that in any 2D array, the second dimension's "folders" each contain exactly the same number of first-dimension "files". For example, in this **[2;4]** array...

```
arTest[ ] = { { "A" ; "B" ; "C" ; "D" } ; { "E" ; "F" ; "G" ; "H" } }
```

...you have **two** folders (the two sets of braces inside the array, not the outer braces) each containing **four** files ("A", "B", and so on). Last time, when we augmented the second dimension of this array, we added three files to the two folders, so that the original **[2;4]** array **{"A";"B";"C";"D"};{"E";"F";"G";"H"}** changed into **{{"A";"B";"C";"D";"I";"J";"K"};{"E";"F";"G";"H";"L";"M";"N"}}**, a **[2;7]** array. This time, however, we're going to keep the same number of files in each folder as the origarray has. What's going to change is the number of folders. Instead of ending up with two folders that each have more files, we'll keep the original number of files in each folder, but we'll end up with more folders. So the number of files in each folder will stay the same, but there will be more total files in the array because the additional folders we've created will obviously contain files (otherwise we wouldn't have created the folders in the first place), and those files will obviously be new.

Therefore, this function call for our new second-dimension function...

```
arTest[ ] := AddElements2Drows ( arOrigarray[ ] ; { { "I" ; "J" ; "K" ; "L" } } )
```

...expands the first dimension's *size* (the rows, or folders) but is actually putting the new values into the *second* dimension (columns, or files). If the first dimension is folders, and that's what we're augmenting, we can't give the second (file) dimension a bigger index. The bottom dimension's index range doesn't go up—it still has 4 items per set. However, in real terms, there'll still be more files because there'll be more sets—more folders. So after this function call is executed, the original **[2,4]** array **{"A";"B";"C";"D"};{"E";"F";"G";"H"}** will now be **{"A";"B";"C";"D"};{"E";"F";"G";"H"};{"I";"J";"K";"L"}**, a **[3;4]** array. We could have added more folders, but each of them needs to have four values. If you try to use more or less than that, the function won't work.

By the way—while no doubt it will seem natural to most folks—we should still pause to note that the first dimension will be expanded at the *end* of its index range. In this example, the new folder holding values I-L was not added at the beginning, in front of the folder holding values A-D; rather, it was added at the end, in back of all the previously existing folders. Thus, if you run the function and then wish to access "K" for some reason, you would type "arTest[3;3]." Typing "arTest[1;3]" would get you "C."

Adding a single row is rather pedestrian, of course, when we could add however many row elements we wish! For this example, though, let's settle for two. This function call doesn't look much different than the ones we used for **AddElements2Dcolumns()**, featuring a duo of the brace-enclosed "sets" of data to add...

```
arTest[ ] := AddElements2Drows
        (
                arTest[ ] ; { { "I" ; "J" ; "K" ; "L" } ; { "M"; "N" ; "O" ; "P" } }
        )
```

...but the key is that we cannot change the number of values per set. We could have had 50 or 100 sets of braces (files/rows) if we wanted, as long as each one held four values (files/columns), since that's the number held in the columns of origarray **`arTest[]`**.

But we're obeying Aristotle's time-tested injunction not to overindulge. As is, this call tells the function to extend the rows by two elements (i.e. add two new folders), and at the end, instead of the original **`[2,4]`** array, we'll have one that's twice as big: **`[4;4]`**. Note the scrupulous maintenance of four pieces of data per set. Once again, at the risk of being repetitive, this is the folder/row function; it can't extend the file/column dimension; therefore, we must religiously specify four pieces of data, no more and no less, for each folder/row we wish to add. In other words, for each element we want to add to the first dimension, we must offer a set of values the number of which is equal to the second dimension's index range.

So how does the function work?

1. As you might expect by now, the first line stays as is except for the new function name.

2. The second line is still the place where we do the restructuring, and it's mostly the same—except for the second param of the restructuring function call, of course. Since we're now focusing on the first dimension of a 2D array, not the second, we give the restructuring function a second parameter consisting not of a 0 and then the number of new values, but rather first the number of new values and then a 0, whereupon the function will not increase the columns but will increase the rows. (The number of new values, of course, is still delivered via the ***`array[0;X]`*** operator):

```
arNew[ ] := ExtenDim ( &arOrigarray[ ]; { arExtra[ 0 ; 1 ] ; 0 } )
```

3. Except for the fact that we have to reverse the two **`arNew[]`** slices, the third line remains the same as it was in the column function:

```
arNew[ arOrigarray[ 0 ; 1 ] + 1 .. ; .. ] := arExtra[ .. ; .. ]
```

Note how the row/folder elements automatically carry the column/file data inside them. We don't have to perform any special magic in **`arExtra[]`**'s first dimension to correctly pass the new values into **`arNew[]`**. Passing the new rows/folders into **`arNew[]`**'s new top slice of space is all it takes. Because the PerfectScript engine sees that top slice, and sees how **`arExtra[]`**'s rows/folders are all being passed into that top slice, it doesn't overwrite **`arNew[]`**'s pre-existing columns/files but instead simply sets the new rows/folders containing the new columns/files *alongside* the pre-existing array contents.

—and then we return the result and we're done:

```
FUNCTION AddElements2Drows ( arOrigarray[ ] ; arExtra[ ] )

        arNew[ ] := ExtenDim ( &arOrigarray[ ]; { arExtra[ 0 ; 1 ] ; 0 } )

        arNew[ arOrigarray[ 0 ; 1 ] + 1 .. ; .. ] := arExtra[ .. ; .. ]

        RETURN( arNew[ ] )
ENDFUNC
```

Adding Values to Both, Old Style

So how to combine these so we don't have two separate functions? In **`ExtenDim()`** we could do both columns and rows at once, but not here. This has to do with the fact that we cannot specify new columns and new rows for the existing array at the same time: remember how scrupulous we had to be about keeping the exact number of rows (folders) the same if we were augmenting columns, or the exact number of values in the columns (the exact number of files) the same if we were augmenting rows? Each of the 2D functions requires that the dimension *not* being augmented be rigorously maintained in its previous form. There's just no way to augment them both at the same time; the mechanisms that make these functions work can't accommodate such a goal. So if we want to augment both rows and columns in the same function, we have to do some logical branching—and that means an IF construction.

The first thing the function itself must do is figure out whether the user wants it to add more columns or more rows . To do this, it will look at the size of the new array. If the new array has the same amount of rows as the origarray, then by process of elimination, the function will know that it's supposed to add new columns (the new array—the one passed to the function as its second parameter—will contain new columns for all the existing rows). On the other hand, if the new array has the same amount of columns as the old array, then the function will do the opposite and add new rows (the new array will contain new rows for all the existing columns).

Our new combinatory function will be called **`AddElements2Dfirst()`**("first" because we'll turn right around and create a second one in just a moment). The extension code for the columns and rows is the same as the second and third lines in **`AddElements2Dcolumns()`** and **`AddElements2Drows()`**. What's different here is the IF code, which compares the origarray with the paramarray (using the ***`array[0;X]`*** operator) to see whether or not their row indices match:

```
        IF (arOrigarray[0;1] = arExtra[0;1])
```

If the row index is the same in each, the function assumes that your column index is different and that you want to add columns, so it executes the immediately subsequent column-adding code that was developed in **`AddElements2Dcolumns()`**. But if the row indices are different, it

assumes that you want to add rows instead and throws you down to the ELSE, which executes the row-adding code from **`AddElements2Drows()`**. So we end up with this:

```
FUNCTION AddElements2D (arOrigarray[ ]; arExtra[ ])

    IF (arOrigarray[0;1] = arExtra[0;1])

        arNew[ ] := ExtenDim (&arOrigarray[ ]; {0; arExtra[0;2]})

        arNew[..;arOrigarray[0;2] + 1..] := arExtra[..;..]

    ELSE

        arNew[ ] := ExtenDim (&arOrigarray[]; {arExtra[0;1]; 0})

        arNew[arOrigarray[0;1] + 1..;..] := arExtra[..;..]

    ENDIF

    Return (arNew[ ])

ENDFUNC
```

...but as we use this function, we need to remember to structure our paramarray appropriately for either rows or columns. The form of the call is utterly crucial. The paramarray must have the same index range as the origarray for at least one or the other dimension; if it doesn't, the macro system generates an error message and stops execution.

The Same-Index-Range False Sense of Security

And what about the other side of the coin—what if the paramarray matches the origarray's index ranges in *both* rows and columns—as in the earlier example of adding rows where a **`[2;4]`** array was turned into a **`[4;4]`** array? The paramarray had to be **`[2;4]`**, and of course the origarray started out that way too. With our function **`AddElements2Dfirst()`**, The paramarray could contain either more columns to add *or* more rows to add. Which is it? Well, as we've seen, the function first checks to see if the *rows* are the same number, so such arrays will be processed as if we are adding more columns, not rows. That's great if you really want to add columns, but if you wanted to add more rows, you're out of luck. We might call this a "same-index-range false sense of security," since the macro is sure that it has done the right thing. Says JDan, "There is really no automatic way to get around this that I can see." You have to use the function twice, doing each augmentation separately. So if we had the **`[2;4]`** array...

```
arOrigarray[ ] = { { "A"; "B"; "C"; "D" } ; { "E"; "F"; "G"; "H" } }
```

...and we wanted to add two more rows (to end up with a **[4;4]** array), we might be tempted to call the function like this...

```
arOrigarray[ ] = AddElements2
(
        arOrigarray[ ] ; { { "I";"J";"K";"L" } ; { "M";"N";"O";"P" } }
)
```

...but the function would treat these as columns and we'd get a **[2;8]** array instead. The workaround is to add each row (with its columnar set) to the array in separate function calls:

```
arOrigarray[ ] = AddElements2 ( arOrigarray[ ] ; { { "I";"J";"K";"L" } } )

arOrigarray[ ] = AddElements2 ( arOrigarray[ ] ; { {"M";"N";"O";"P" } } )
```

This is clumsy, but it gets you past the same-index-range false-security problem (instead of a **[2;4]** paramarray, we're passing it a **[1;4]** array).

Adding Values to Both, New Style

Now a new issue: in the new IF-ELSE version of our function, the second param of **ExtenDim()** is pretty much alike in each call, differing not in kind but merely in order. One of the sizes to expand by is a 0, and the other is the size of the new array in the other dimension, or vice versa. Since what differs is only the order in which these types of size specifications appear, we can consolidate this so that we only call **ExtenDim()** once in the function. We'll do this by using a **FORNEXT** loop to build a new one-dimensional array that contains an index range (total number of elements) equal to the number of dimensions we want to add, and in which each of the elements contains a numerical value that equals the number of elements by which we want to extend the corresponding dimension of the origarray.

That sentence is a mouthful, but once you plow through it a few times, you'll see that it describes precisely the kind of array that **ExtenDim()** requires in order to do its restructuring. let's expand an example briefly adduced earlier, a **[3;2]** array—**{{"A";"B"};{"C";"D"};{"E";"F"}}**—in which we ultimately want five columnar values, ending up with, say, **{{"A";"B";"AA";"BB";"AB"};{"C";"D";"CC";"DD";"CD"};{"E";"F";"EE";"FF";"EF"}}**, a **[3;5]** array. In that case, we're going to have to feed **ExtenDim()** a one-dimensional array that has a zero for the rows and a three for the columns—so if our **FORNEXT** loop builds a **[0;3]** array, we can then slip this new array into **ExtenDim()**'s second parameter and take it out of the **IF** statement. No matter whether it's rows or columns, our new "resizing array"—let's call it "arResize[]"—will allow the function to handle it because the **FORNEXT** will automatically build **ExtenDim()**'s paramarray to the correct specifications.

Or it will as long as we're mindful of the same-index-range false-security problem. That issue still lurks in the background; all the new **FORNEXT** code will do is take **ExtenDim()** out of the

IF-ELSE, reducing the number of times it appears in the function. Multiple instances of the function will still need to be called if the paramarray yields the exact same dimensions as the origarray.

Here's the entire function. Since it'll be the final function for 2D arrays, we'll simply call it "AddElements2D":

```
FUNCTION AddElements2D (arOrigarray[ ]; arExtra[ ])
        declare arResize[ arOrigarray[0;-1] ]
        fornext (i; 1; arResize[0])
                if (arOrigarray[0;i] = arExtra[0;i])
                        arResize[ i ] := 0
                else
                        arResize[ i ] := arExtra[0;i]
                endif
        endfor
        if (arResize[ ] = 0)
                prompt ("Equal dimensions prevent array resizing")
                pause
                return ( { 0  } )
        endif
        n = 0
        fornext (i; 1; arResize[0])
                if (arResize[ i ] != 0)
                        n := n  +1
                endif
        endfor
        if (n > 1)
                prompt ("Cannot expand array by multiple dimensions at one time")
                pause
                return ({0})
        endif
        arNew[ ] = ExtenDim (arOrigarray[ ]; arResize[ ])
```

```
    if (arOrigarray[0;1] = arExtra[0;1])
        arNew[..;arOrigarray[0;2] + 1..] := arExtra[..;..]
    else
        arNew[arOrigarray[0;1] + 1..;..] := arExtra[..;..]
    endif
    RETURN(arNew[ ])
ENDFUNC
```

Two-Dimensional Add-on Walkthrough

1. The first line remains the same, with the origarray and the paramarray...

```
    FUNCTION AddElements2 (arOrigarray[ ]; arExtra[ ])
```

...but as always, remember to structure the paramarray correctly to achieve the results you desire.

2. In the second line, we declare our new resizing array:

```
    declare arResize[arOrigarray[0;-1] ]
```

Because it's ultimately going into **`ExtenDim()`**'s second param, which must always be a 1D array, we make this a 1D array right from the start. Its index range is the number of dimensions in the origarray, which you'll remember is exactly what **`ExtenDim()`** requires: the **`ExtenDim()`** parameter's index range must be the same as the origarray's total dimensions (whereas it's the actual values in each of those elements that determines precisely how much to extend each dimension). In this function, **`arOrigarray[0;i]`** will always be 2, since the routine is designed only for 2D arrays. In fact, the function could be hard-coded so that the second line reads **`declare arResize[2]`**, but eventually we want to end up with a function for extending and adding values to an array of any size, and for that purpose we need the **`arOrigarray[0;i]`** syntax, which will not restrict us solely to a 2D array but allow us to handle all arrays. So we might as well start out with it here.

OK, after the second line is finished, **`arResize[]`** exists and has the right number of dimensions (two, in the current function). There are no elements in either dimension as yet, however, let alone the values that are supposed to tell **`ExtenDim()`** how many elements to add to each dimension. Nope, right now **`arResize[]`** is totally empty. But that's about to change.

3. The third line starts a **`FORNEXT`** loop that will add elements and values to the resizing array:

```
    fornext (i; 1; arResize[0])
```

Once again, **`ExtenDim()`** works by way of a unidimensional array whose index range—the number of elements it has—equals the number of dimensions in the origarray, but in which the

actual values of those elements denote the number of elements, respectively, by which each of those dimensions is to be extended (for more on the second param of **ExtenDim()**, see "4D and Beyond | The New Call," above). The loop uses both a standard start value and increment value of 1, so there's nothing out of the ordinary there; more noteworthy is the stop value, which isn't hard-coded but is instead soft-coded to be the total number of elements in the just-declared resizing array—which, of course, is also the total number of dimensions from the origarray (assuming we made a correct function call). So the loop will go around as many times as there are dimensions in the origarray—that is to say, in this function, twice, since it's limited to 2D arrays. (If you pass this function an array with more or less dimensions than two, the PerfectScript engine will generate an error.)

Getting Loopy

4a. Each time the loop iterates, the **IF** fires. The first thing that happens is a test using the ***array*[0;*X*]** operator: do the origarray and the paramarray have the same index range in the dimension denoted by the current iteration number?

```
if (arOrigarray[ 0;i ] = arExtra[ 0;i ])
```

Thus on the first iteration the test will operate on the first dimension of the array; at that point, the test **if (arOrigarray[0;i] = arExtra[0;i])** will actually be "arOrigarray[0 ; 1] = arExtra[0 ; 1]" under the hood, and the [0;1] special array operator returns the index range of its array's first dimension. On the second iteration, the loop will test the second dimension; and if the routine could handle other-dimensional arrays, the third iteration would test the third dimension, the fourth the fourth, and so on. But in this array where we're still limited to 2D arrays, the first iteration will always test rows (since rows constitute the first dimension), the second iteration will test columns (the second and last dimension), and then the loop will terminate since its stop value equals the number of dimensions in the origarray, which were given to **arResize[]**.

4b. What happens inside the **IF** depends on how the test worked out.

- If the respective arrays indeed have equal-sized dimensions in the slot being tested (first and second, in this function), the **IF** code runs. If the rows are the same, they answer the **IF** test correctly the first time through the loop, causing the **IF** statement to fire...

  ```
  arResize[ i ] := 0
  ```

  ...which gives the first element in **arResize[]** a zero because we obviously don't want to extend that dimension (if we did want to do that, we wouldn't have given **arExtra[]** the same number of rows as the origarray when we called the function; for more on this issue, see **The Same-Index-Range False Sense of Security, Reprise**, below).

  The second time through, the test registers negative and the **ELSE** code runs...

  ```
  arResize[ i ] := arExtra[0;i]
  ```

...which gives the second element in **arResize[]** a number equal to the number of elements in **arExtra[]**'s second dimension. So in this function, **arResize[2]** gets whatever number the macro writer passed into **arExtra[]**'s columns, and that's the number of slots (elements) which will be added to the columns.

- If the first dimensions in the two arrays *don't* share the same size, the routine assumes that we want to extend these dimensions (here, the rows) and gives control to the **ELSE** on the first iteration, so...

```
arResize[ i ] := arExtra[0;i]
```

...gives **arResize[1]** a numeric value which is the same as the quantity of elements that **arExtra[]** has in its first dimension (i.e. how many rows it has). That's how many row slots will be added in **ExtendDim()**. So if we gave **arExtra[]** the paramarray **{{"AA";"BB";"AB"};{"CC";"DD";"CD"};{"EE";"FF";"EF"}}** at function call, here **arResize[1]** would get a value of **3**.

During the second iteration, the test comes up positive, so the IF fires off its own statement...

```
arResize[ i ] := 0
```

...and the second element of **arResize[]** gets a value of **0**. Columns will not be resized!

This can be confusing in the abstract, so let's work through an example. Let's say that we have **arTest[]={{"A";"B"};{"C";"D"}}** and we want to extend columns by 5, so our paramarray in the function call was **{{"E";"F";"G";"H";"I"};{"J";"K";"L";"M";"N"}}**. And, of course, since this function is only meant to handle arrays of two dimensions, we didn't try any monkey business with extra braces—just the single outer pair that doesn't count because it merely says, "Hey, array here," and the individual, equal-level inner pairs of braces that determine how many rows (first dimensions) there are. So that's not an issue—when the loop fires off this **IF**, the test **arOrigarray[0;i] = arExtra[0;i]** is straightforward. The first time, when **i** is 1, the first dimension (i.e. the rows) is the object of the test, so the resulting equation is really "2 = 2" (since both arrays have two rows), so the **IF** registers "True" whereupon the fifth line fires and the passes a value of zero to the element of the resizing array denoted by the iteration number. In this case, **arResize[i]** is actually "arResize[1]," and that element (which until now was completely empty) gets a numerical value of 0.

Note that this element is *1*, the *first*. This is critical to what we're trying to do—extend columns. In **ExtenDim()**, whatever's not getting extended must be denoted by zero, and this operation gave us a nice fat zero for just that purpose.

Then the loop fires one more time. Now **i** is 2, so the column index rather than the row index is presently being tested. Given our current example, the **IF**'s test expression works out to 2=5. The

result is "False," so the macro skips down to **ELSE** and makes the second element in the unidimensional resizing array however large the paramarray index range is—**5**, in this case.

At this point the loop terminates; its stop value was the total number of dimensions in the origarray, and in this function, since we're only passing in 2D arrays, that means the loop terminates at 2. But the job has been done; **arResize[]** now equals **{0; 2}**, and the array is ready to serve as the second parameter in our old friend **ExtenDim()**.

The Same-Index-Range False Sense of Security, Reprise

Stop right there! Some readers may have noticed a problem. What happens if the origarray and paramarray have the same number of rows and the same number of columns, respectively? The **IF** statement will run each time, the **ELSE** will never run, and both elements of **arResize[]** will end up with zeroes for their values. **ExtenDim()** will run but won't end up doing anything so the origarray won't get resized, and when the macro gets down to the addition code it will generate an error because it'll be passing extra values to **arNew[]** for which that worthy array has no space.

In one sense this same-index-range problem is good because it forces us to be more careful coders in the main macro, calling **AddElements2()** twice and precisely controlling its behavior rather than running the risk of extending columns when we really want to extend rows or vice versa, as could have happened with the first, "old-style" function that didn't have a loop. But still, it's poor coding practice to leave users at the mercy of default error messages, so let's do some error-checking to prevent such unhappy occurrences:

```
if (arResize[ ] = 0)

        prompt ("Equal dimensions prevent array resizing")

        pause

        return ( { 0 } )

endif
```

Since an array with nothing but numeric zeroes in its elements will return a zero when queried, we can use an **IF** with the simple test-expression **arResize[ ] = 0** and pop up a normal prompt box. Then we forcibly terminate the function by using the **return** command. (All functions must return something, and returning is also the very last thing they do, so if you force them to return at any given point, the function stops—exactly what we want if the user has committed the same-index-range sin.)

Here, however, we can't merely use "return" all by itself. This macro passes the function's return value into an array, so that return value must be an array or the macro will crash with an error message that says something like "You tried to assign a non-array value to an array variable."

`return ({0})` neatly solves the problem by returning not just an array but also pretty much the simplest array possible, one unlikely to ever be a valid return value (so that other parts of the macro, outside the function, won't be able to use it and thereby generate incorrect but syntactically valid results).

Overzealous Expansion

But we're not out of the woods yet! There's another potential mistake lurking in the macrojungle, waiting to pounce: there's no logical way of extending (nor of adding new values) to more than one dimension at a time. If we try to do that by passing a value higher than zero to each dimension of the **`arExtra[]`** parameter, only one of the dimensions will be resized by **`ExtenDim()`** and the macro will either add values to that dimension but leave the other untouched, puzzling the user with an unaugmented array; or the macro will fail to augment either dimension, because the function's last **`IF`** (see below) will return **`False`**.

To alert users to this problem when it arises, we'll make sure that only a single element of **`arResize[]`** is larger than 0 before running the augmentation code. This requires setting a counter variable **`n`** and then using it in a **`FORNEXT`** loop before our prompt...

```
n = 0
fornext (i; 1; arResize[0])
        if (arResize[ i ] != 0)
                n := n  +1
        endif
endfor
if (n > 1)
        prompt ("Cannot expand array by multiple dimensions at one time")
        pause
        return ({0})
endif
```

...but though that looks like a lot of lines for just a single coding goal, it's still a fairly simple sequence withal. You just run the loop as many times as there are dimensions, and during each iteration, the macro checks to see whether that particular dimension has a number of elements larger than 0. Each time there are more than 0 elements, the counter goes up by 1. Then after the loop is finished, the standalone **`IF`** underneath it checks to see if the counter is more than 1. It'll only meet that test if **`n`** was augmented in the **`FORNEXT`** because neither of **`arResize[]`**'s dimensions were zero (which would only occur if our function call made both of **`arExtra[]`**'s dimensions a different size than those of the origarray). If that happened, the prompt goes off and alerts the user to the problem.

Done Deal

If our function call is careful and we safely pass the guardian error-checking sequences, our origarray will be extended...

```
arNew[ ] = ExtenDim (arOrigarray[ ]; arResize[ ])
```

...and in the form of **`arNew[]`**, it is now ready to be augmented with gen-yoo-wine values. Now for the first dimension we just repeat the "equality test," which already proved its usefulness during the resizing code...

```
if (arOrigarray[0;1] = arExtra[0;1])
```

...and if the rows equal each other in number, they're not supposed to be augmented, so pass the entire paramarray into all the origarray's rows but only the top slice of its columns:

```
arNew[..; arOrigarray[0;2] + 1..] := arExtra[..;..]
```

Conversely, if the rows didn't equal each other, then we *do* want to augment them, so we pass the paramarray into the top (unoccupied) slice of its rows, bringing the columns along for the ride...

```
arNew[arOrigarray[0;1]+1.. ;..] := arExtra[..;..]
```

...and either way we've ended up with a correctly augmented array.

Adding Values to All

Now it's time to go for broke, creating code that will not only augment arrays of all allowable dimensions but also better address the "same-index-range false-security" issue. What if, in cases where the function's normal logic doesn't determine which dimension to augment, we could simply *tell* the function which dimension to extend so that it wouldn't give all the dimensions a 0?

That's exactly what we'll do. At the spot where we display the "Equal dimensions prevent array resizing" message, we'll add new commands that make **`arResize[x]`** contain a number that matches the quantity of rows in the new array, adding one additional parameter, called **`m`**, which will be the dimension number that the array should be expanded by (though in order to conserve system resources, we'll bring it into play only if the rows and columns are both equal). We'll still retain the "resizing" error message just in case, but it'll now be in much less danger of actually showing up.

Function Walkthrough

The function is a long one, so let's look at it in sections—along with a running example—before reproducing the whole thing. The first line of the function, the name and parameters, is the same as before except for the new **`m`** parameter:

```
FUNCTION AddElements(arOrigarray[ ]; arExtra[ ]; m)
```

For our example, let's say that the array we want to augment, **`arTest[]`** out in the (imaginary) main macro, is a 3D array with a **`[2;3;1]`** structure like so...

```
{
        { {"A"};{"B"};{"C"} }  { {"D"};{"E"} {"F"} }
}
```

...and that we want it to have one more plane, or third-dimension "folder," adding values G through I as the "files" in the first dimension (columns). This requires a **`[1;3;1]`** paramarray—all dimensions matching the origarray's except for the dimension we want to augment (the third)—and the function call would therefore go like this (although remember that we need not pass **`arExtra[]`**'s value as an array literal):

```
arTest[ ] := AddElements(arTest[ ]  ;  {{{"G"};{"H"};{"I"}}}  ;  3)
```

Creating `ExtenDim[]`'s Temparray: `arResize[]`

When we first get inside the function, the code remains unchanged from that of **`AddElements2D()`**; if you'd like a detailed reminder of how it works, feel free to go back and review that explanation. It's the portion where we declare the one-dimensional **`arResize[]`** and automatically give zeroes to its elements except for the one whose index corresponds to the dimension that's supposed to be extended. In our example here, **`arResize[]`** is declared a one-dimensional array with three element slots. **`arResize[]`** then dives into the **`FORNEXT`**, where it goes around three times. The first time, the **`IF`** tests False, so the **`ELSE`** fires and **`arResize[1]`**, which corresponds to the "plane" dimension of the origarray, gets a value of **`1`**. The **`IF`** tests positive the next two times through the loop, so the second and third elements (corresponding to the rows and columns of the origarray) , both get values of **`0`**.

```
declare arResize[ arOrigarray[0;-1] ]

fornext (i; 1; arResize[0])

        if (arOrigarray[0;i] = arExtra[0;i])

                arResize[ i ] := 0

        else

                arResize[ i ] := arExtra[0;i]

        endif

endfor
```

Error-Checking and Forced Resizing with M

Now we hit the new-and-improved same-index section, where the new parameter **m** comes into its own. It won't fire for us, however, because our function call was completely kosher. The **IF** will test negative and kick **arResize[]** on down past the **ENDIF** into the next part of the function, and that will be that. But if we had wanted to extend the third dimension of **arTest[]** by three instead of by one, this part would save our bacon. Previously, that would have been impossible; the fact that **arTest[]** itself already had three planes meant that we would have had to make two function calls, one to expand it to two planes and another for the third. But with paramater **m** holding **3**, we no longer need to do that. Let's step through this in detail.

1. First, the IF tests to see whether **arResize[]** equals zero. If we were trying to add three more planes, that would be the case because all three dimensions would have triggered values of zero for the corresponding elements of **arTest[]** (in the **FORNEXT** directly above)...

```
if (arResize[ ] = 0)
```

2. ...and the next **IF** would run its test. This tests for not one but two conditions, neither of which is likely, but hey, we're covering all the bases. First, it's conceivable that a clueless macro writer might have actually passed our new parameter **m** a value of **0**, in which case even **m** won't do us any good; second, the same clueless coder might have misconstructed their paramarray so that it has less dimensions than **m**'s value, or just mistakenly given **m** a higher value by accident, two other scenarios in which **m** won't do us any good...

```
if ( (m = 0 ) or ( m  > arResize[0] ) )
```

...and in both situations the macro would generate an error and shut down with one of those cryptic messages if we didn't stop it here with a better one. So despite our best efforts we're backed into a corner and we pop up an error message (different from the previous one because the kinds of errors that can cause it are different):

```
        prompt ("Dimensional mistake prevents array resizing")
        pause
        return ( { 0 } )
endif
```

3. However, these problems shouldn't come up (*you'll* never write any code that would cause them, hmm?); the situation should be the one that **m** was designed for. In that case, the macro gets the index range of the paramarray dimension that matches **m**; i.e. if **m** holds a value of **3**, the macro takes the index range of **arExtra[]**'s third dimension. This index range is then passed to the corresponding element in **arResize[]**...

```
            arResize[m] := arExtra[0; m]
        endif
```

...and the **ENDIF** closes the section off by delivering an **arResize[]** that is no longer 0 and is on its way to serving nicely as **ExtenDim[]**'s paramarray.

Error-Checking for Overeager Augmentation

But hasn't yet made it to **ExtenDim[]**. Now comes the section where we check to see whether the user tried to augment two (or more) dimensions at the same time, and only after that do we call **ExtenDim()**. We don't have fancier code for any of this, so the section remains the same as that in **AddElements2D()** (go back to the relevant section for a detailed technical review). Given our **[2;3;1]** origarray (which, remember, we wish to augment with **{{{"G"};{"H"};{"I"}}}**), the resizing array **arResize[]** now has an index range of **3** with a visual layout of **{1;0;0}**. The **FORNEXT** therefore iterates three times (because the index range of **arResize[]** is its stop value) but **n** gets increased only once (because only one of **arResize[]**'s elements has a value that's not zero)...

```
n := 0

fornext (i; 1; arResize[0] )

    if (arResize[i] != 0)

        n := n + 1

    endif
endfor
```

...so after the loop, **if (n > 1)** returns False, the prompt message remains dormant, **arResize[]** slips into **ExtenDim[]**—no doubt with a sigh of relief, after all that error-checking—to do the job it was always intended for, and **arNew[]** is finally created with the origarray's values and the paramarray's new structure (one new plane, or top-level "folder"):

```
if (n > 1)
    prompt ("Cannot expand array by multiple dimensions at one time")
    pause
    return ( { 0 } )
endif

arNew[ ] = ReDimN (&arOrigarray[ ]; arResize[ ])
```

3, 2, 1—Augment!

Now we're ready to actually add values to our extended array, and we've got new code so that no matter how many dimensions we have (subject to PerfectScript's maximum of ten), the function will take care of them. The new function does this by eliminating the **IF** used in **AddElements2D[]** to discriminate between types of augmentation (rows or columns), since that test was predicated on the assumption that the function would only be asked to augment an array of two dimensions. It won't work when there are any more than two choices—yet we want ten! The function obviously needs a different mechanism for making the correct value assignment to the newly extended array.

It's a tough job. The task at hand is to fill up the new slice (by which **ExtenDim()** just extended the origarray in **arNew[]**) with the values we passed to **arExtra[]** at function call: the new plane with its three new rows each containing a new columnar value (or a top-level folder containing three bottom-level folders which each have one file), i.e. **{{{"G"};{"H"};{"I"}}}**. To pass this structure and values into that top slice of the newly restructured array, we have to give **arNew[]** the correct slice syntax—which in our current case will be **arNew[3..;..;..]**. But the function has no way of magically knowing that; it has to provide for all possibilities and then just see which one turns up. It has to have a way of being ready for a top slice from any of the dimensions—what if we had wanted to augment the first dimension instead, and the syntax should be **arNew[..;..;3..]**?

In addition, all the dimensions that *aren't* being augmented require a normal "total" slice (from the beginning of the dimension to the end), the **[..]**. Somehow the function has to make sure that all this happens, and that it happens for all the dimensions in each allowable range of dimensions, from one to ten..

JDan's ten-choice solution involves a close logical appreciation of slices and a re-use of **arResize[]**, which is now just hanging around in memory, wasting system resources. We might as well recycle it in a **FORNEXT**. Here's what the loop will do: remember how each element of **arResize[]** that *doesn't* correspond to an augmented dimension has a value of zero, because that's what **ExtenDim[]** requires? Well, we're going to change that. Now, each of those non-augmented dimensions will get a value of **1**.

And it gets even weirder. The **arResize[]** element corresponding to the dimension that *is* getting augmented will lose its value of **1**, instead being overwritten to contain a value equal to the origarray's final element in that dimension, plus one:

```
fornext (i; 1; arResize[0])

        if (arResize[ i ] = 0)
                arResize[ i ] := 1
        else
                arResize[ i ] := arOrigarray[0;i] + 1
        endif
```

```
    endfor
```

Recall that in our running example, **arResize[]** is **{1;0;0}**. The first iteration therefore tests negative because **arResize[1]** isn't equal to zero (it currently has a value of **1**). This result triggers the **ELSE**, which promptly gives **arResize[1]** a new value equivalent to the last element of the origarray's first dimension, plus one. Since our example origarray has a dimensional structure of **[2;3;1]**, the **ELSE** gives **arResize[1]** a value of 2 + 1 (i.e. **3**).

The next two iterations of the loop each test positive, giving both **arResize[2]** and **arResize[3]** values of **1**.

Huh? Plus one, then one and one? What's going on here?

The mystery becomes a bit clearer when we recall that the last element of the origarray's to-be-augmented dimension, *plus one*, matches the first index of the newly added, still-empty top slice of **arNew[]**. That's what we have to pass the paramarray values into. And the value of **1** matches the first index in any array. So by the time we're done with the **FORNEXT**, **arResize[]** has the first values of all the necessary **arNew[]** slices, both augmented and unaugmented. The next step will be to use those values to create those slices.

The origarray-dimension-plus-one method is familiar; we've been using it since the very first function for adding values, the basic one-dimensional version. Using **1** in slices for regular, unextended, unaugmented dimensions is what's new here. But we won't be adding it. We'll simply be using it as the first index range in the unaugmented dimensions' slices, like so: **[..]**. True, we've never used **1** before in the slices; aside from the augmented dimension's top slice, we've always just used the barebones **[..]**. But **[1..]** is exactly the same thing, so it'll achieve the same result, and it has to be done in order to formulate the correct **arNew[]** slices for the augmented dimension as well. We've got two different kinds of slices to accomodate, and this is the most efficient way to do it.

So the **FORNEXT** does part of this work by overwriting **arResize[]** with the first values of the assignment slices. The actual assignment code, a **SWITCH**, takes those values and does the rest. Thus, the **FORNEXT** and the assignment **SWITCH** work cooperatively and don't make sense apart from each other, so keeping in mind that our example resizing array **arResize[]** is now **{3;1;1}**, let's look at the **SWITCH** (reproducing the first three **CaseOf**s only, in the interests of space):

```
switch (arOrigarray[0;-1])

    caseof (1):
        arNew[arResize[1]..] := arExtra[..]

    caseof (2):
        arNew[arResize[1]..;arResize[2]..] := arExtra[..;..]
```

```
caseof (3):
        arNew[arResize[1]..;arResize[2]..;arResize[3]..] := arExtra[..;..;..]
```

The paramarray **arExtra[]** remains the same; as before, **arExtra[]** holds the values we want to pass into **arNew[]**, and we still use the simple default **[..]** beginning-to-end-of-dimension syntax for **arExtra[]**. It's the slices in **arNew[]** that use the new **arResize[]** we've been talking about. By using the elements of **arResize[]** in linear order (1, 2, 3, etc.) as the beginning elements in **arNew[]**'s slices, the function will always have the correct top slice in the right dimensional slot, and all the other dimensions will have—as they should—normal beginning-to-end-of-dimension slices.

In our example, we're trying to augment the third dimension, so that's the one that has to specify the top slice with extended elements; the others should be normal, full default slices. And that's precisely the result achieved for us by **arResize[]**. After the **FORNEXT**, it's **{3;1;1}**. So when the macro hits **switch (arOrigarray[0;-1])**, registers **3**, and then jumps down to the third **CaseOf** , **arNew[]** is waiting with correctly open arms: **arResize[1]** is 3, **arResize[2]** is 1, and **arResize[3]** is 1. Therefore the code on the left of the assignment operator is actually **arNew[3..; 1..; 1..]**, exactly what **arExtra[]** needs to successfully pass its values into **arNew[]**.

And that's it. Those are the principles by which the final augmentation function works. Below you'll find the whole thing, unformatted for easy cutting-and-pasting into WordPerfect macros. If you want to delve more deeply into the topic of array manipulation, here's the relevant thread on WordPerfect Universe from which the dimension-extension and -augmentation code in this document was taken: http://wpuniverse.com/vb/showthread.php?s=835740a1faeb006c8cfaaf0137346377&threadid=2374. In addition to the code reproduced here, that thread offers a solution to the WP8 single-element-array bug as well as a nifty function for completely re-dimensioning arrays rather than merely resizing and augmenting existing dimensions.

And you have now earned your M.A. degree—Master of Arrays! Go forth and conquer Macroland!

// WP8 users should remember to remove all passage by reference from both this function and **ExtenDim()**:

```
FUNCTION AddElements (arOrigarray[]; arExtra[]; m)

        declare arResize[arOrigarray[0;-1]]

        fornext (i; 1; arResize[0])

                if (arOrigarray[0;i] = arExtra[0;i])

                        arResize[i] := 0
```

```
        else
                arResize[i] := arExtra[0;i]
        endif
endfor
if (arResize[] = 0)
        if ((m = 0) or (m > arResize[0]))
                prompt ("Dimensional mistake prevents array resizing")
                pause
                return ({0})
        endif
        arResize[m] := arExtra[0;m]
endif
n := 0
fornext (i; 1; arResize[0])
        if (arResize[i] != 0)
                n := n + 1
        endif
endfor
if (n > 1)
        prompt ("Cannot expand the array in more than one dimension at a time")
        pause
        return ({0})
endif
```

```
arNew[] := ExtenDim (&arOrigarray[]; arResize[])

fornext (i; 1; arResize[0])

        if (arResize[i] = 0)

                arResize[i] := 1

        else

                arResize[i] := arOrigarray[0;i]+1

        endif

endfor

switch (arOrigarray[0;-1])

caseof (1):
        arNew[arResize[1]..] := arExtra[..]

caseof (2):
        arNew[arResize[1]..;arResize[2]..] := arExtra[..;..]

caseof (3):
        arNew[arResize[1]..;arResize[2]..;arResize[3]..] := arExtra[..;..;..]

caseof (4):
        arNew[arResize[1]..;arResize[2]..;arResize[3]..;arResize[4]..] := arExtra[..;..;..;..]

caseof(5):
        arNew[arResize[1]..;arResize[2]..;arResize[3]..;arResize[4]..;arResize[5]..] :=
        arExtra[..;..;..;..;..]

caseof(6):
        arNew[arResize[1]..;arResize[2]..;arResize[3]..;arResize[4]..;arResize[5]..;arResiz
        e[6]..] := arExtra[..;..;..;..;..;..]

caseof(7):
        arNew[arResize[1]..;arResize[2]..;arResize[3]..;arResize[4]..;arResize[5]..;arResiz
        e[6]..;arResize[7]..] := arExtra[..;..;..;..;..;..;..]

caseof(8):
        arNew[arResize[1]..;arResize[2]..;arResize[3]..;arResize[4]..;arResize[5]..;arResiz
        e[6]..;arResize[7]..;arResize[8]..] := arExtra[..;..;..;..;..;..;..;..]
```

```
        caseof(9):
            arNew[arResize[1]..;arResize[2]..;arResize[3]..;arResize[4]..;arResize[5]..;arResiz
            e[6]..;arResize[7]..;arResize[8]..;arResize[9]..] := arExtra[..;..;..;..;..;..;..;..;..]

        caseof(10):
            arNew[arResize[1]..;arResize[2]..;arResize[3]..;arResize[4]..;arResize[5]..;arResiz
            e[6]..;arResize[7]..;arResize[8]..;arResize[9]..;arResize[10]..] :=
            arExtra[..;..;..;..;..;..;..;..;..;..]

        default:

        endswitch

        RETURN(arNew[])

ENDFUNC
```