Reference Counting Basics¶

A PHP variable is stored in a container called a "zval". A zval container contains, besides the variable's type and value, two additional bits of information. The first is called "is_ref" and is a boolean value indicating whether or not the variable is part of a "reference set". With this bit, PHP's engine knows how to differentiate between normal variables and references. Since PHP allows user-land references, as created by the & operator, a zval container also has an internal reference counting mechanism to optimize memory usage. This second piece of additional information, called "refcount", contains how many variable names (also called symbols) point to this one zval container. All symbols are stored in a symbol table, of which there is one per scope. There is a scope for the main script (i.e., the one requested through the browser), as well as one for every function or method.

A zval container is created when a new variable is created with a constant value, such as:

Example #1 Creating a new zval container

```
<?php
$a = "new string";
?>
```

In this case, the new symbol name, `a`, is created in the current scope, and a new variable container is created with the type string and the value `new string`. The "is_ref" bit is by default set to **`false`** because no user-land reference has been created. The "refcount" is set to `1` as there is only one symbol that makes use of this variable container. Note that if "refcount" is `1`, "is_ref" is always **`false`**. If you have » Xdebug installed, you can display this information by calling **xdebug_debug_zval()**.

Example #2 Displaying zval information

```
<?php
$a = "new string";
xdebug_debug_zval('a');
?>
```

The above example will output:

```
a: (refcount=1, is_ref=0)='new string'
```

Assigning this variable to another variable name will increase the refcount.

Example #3 Increasing refcount of a zval

```
<?php
$a = "new string";
$b = $a;
xdebug_debug_zval( 'a' );
?>
```

The above example will output:

```
a: (refcount=2, is_ref=0)='new string'
```

The refcount is `2` here, because the same variable container is linked with both *a* and *b*. PHP is smart enough not to copy the actual variable container when it is not necessary. Variable containers get destroyed when the "refcount" reaches zero. The "refcount" gets decreased by one when any symbol linked to the variable container leaves the scope (e.g. when the function ends) or when a symbol is unassigned (e.g. by calling unset()). The following example shows this:

Example #4 Decreasing zval refcount

```
<?php
$a = "new string";
$c = $b = $a;
xdebug_debug_zval( 'a' );
$b = 42;
xdebug_debug_zval( 'a' );
unset( $c );
xdebug_debug_zval( 'a' );
?>
```

The above example will output:

```
a: (refcount=3, is_ref=0)='new string'
a: (refcount=2, is_ref=0)='new string'
a: (refcount=1, is_ref=0)='new string'
```

If we now call `unset($a);`, the variable container, including the type and value, will be removed from memory.

Compound Types¶

Things get a tad more complex with compound types such as arrays and objects. As opposed to scalar values, arrays and objects store their properties in a symbol table of their own. This means that the following example creates three zval containers:

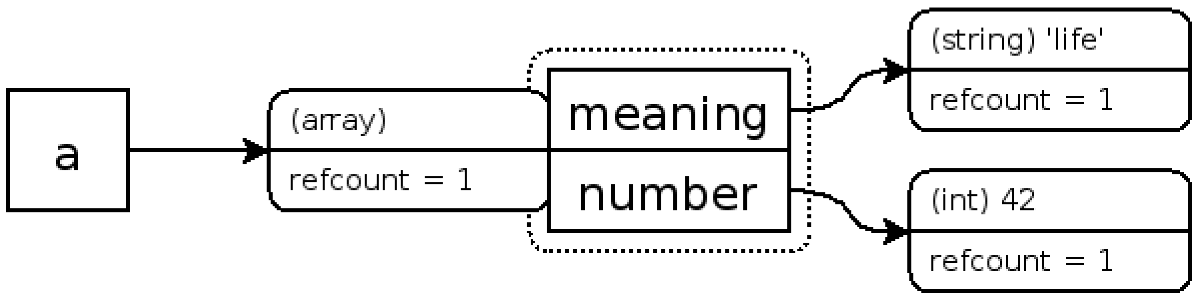
Example #5 Creating a array zval

```
<?php
$a = array( 'meaning' => 'life', 'number' => 42 );
xdebug_debug_zval( 'a' );
?>
```

The above example will output something similar to:

```
a: (refcount=1, is_ref=0)=array (
   'meaning' => (refcount=1, is_ref=0)='life',
   'number' => (refcount=1, is_ref=0)=42
)
```

Or graphically



The three zval containers are: *a*, *meaning*, and *number*. Similar rules apply for increasing and decreasing "refcounts". Below, we add another element to the array, and set its value to the contents of an already existing element:

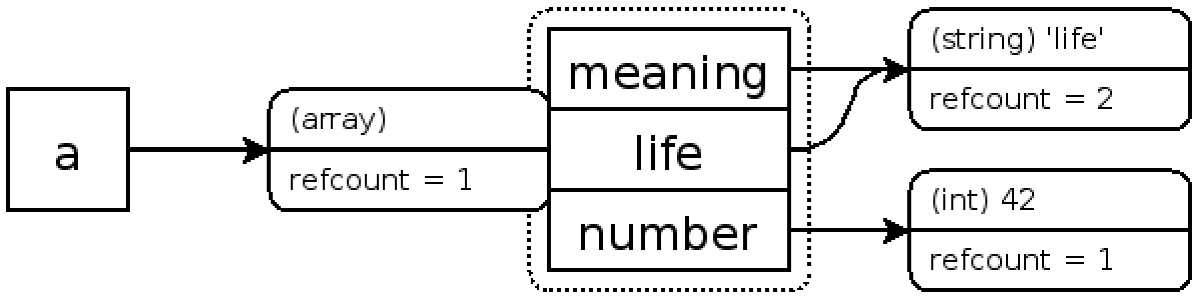
Example #6 Adding already existing element to an array

```
<?php
$a = array( 'meaning' => 'life', 'number' => 42 );
$a['life'] = $a['meaning'];
xdebug_debug_zval( 'a' );
?>
```

The above example will output something similar to:

```
a: (refcount=1, is_ref=0)=array (
   'meaning' => (refcount=2, is_ref=0)='life',
   'number' => (refcount=1, is_ref=0)=42,
   'life' => (refcount=2, is_ref=0)='life'
)
```

Or graphically



From the above Xdebug output, we see that both the old and new array elements now point to a zval container whose "refcount" is `2`. Although Xdebug's output shows two zval containers with value `'life'`, they are the same one. The **xdebug_debug_zval()** function does not show this, but you could see it by also displaying the memory pointer.

Removing an element from the array is like removing a symbol from a scope. By doing so, the "refcount" of a container that an array element points to is decreased. Again, when the "refcount" reaches zero, the variable container is removed from memory. Again, an example to show this:

Example #7 Removing an element from an array

```
<?php
$a = array( 'meaning' => 'life', 'number' => 42 );
$a['life'] = $a['meaning'];
unset( $a['meaning'], $a['number'] );
xdebug_debug_zval( 'a' );
?>
```

The above example will output something similar to:

```
a: (refcount=1, is_ref=0)=array (
   'life' => (refcount=1, is_ref=0)='life'
)
```

Now, things get interesting if we add the array itself as an element of the array, which we do in the next example, in which we also sneak in a reference operator, since otherwise PHP would create a copy:

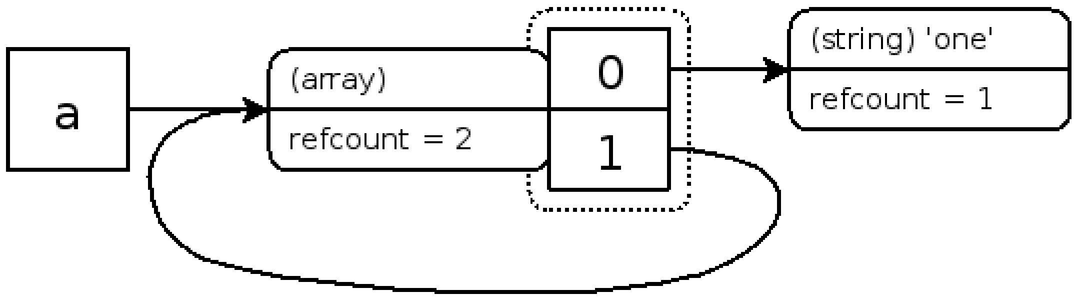
Example #8 Adding the array itself as an element of it self

```
<?php
$a = array( 'one' );
$a[] =& $a;
xdebug_debug_zval( 'a' );
?>
```

The above example will output something similar to:

```
a: (refcount=2, is_ref=1)=array (
   0 => (refcount=1, is_ref=0)='one',
   1 => (refcount=2, is_ref=1)=...
)
```

Or graphically



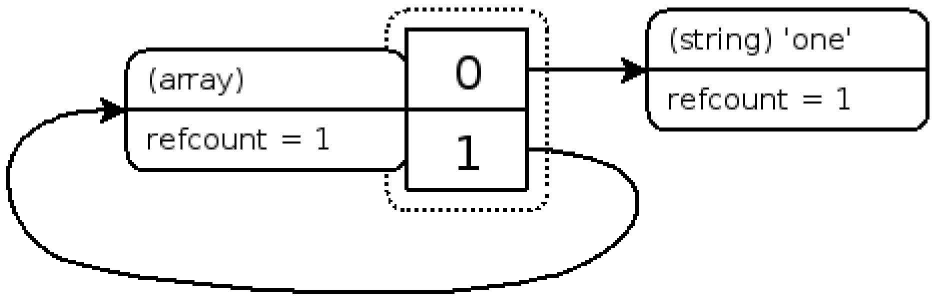
You can see that the array variable (*a*) as well as the second element (*1*) now point to a variable container that has a "refcount" of `2`. The "..." in the display above shows that there is recursion involved, which, of course, in this case means that the "..." points back to the original array.

Just like before, unsetting a variable removes the symbol, and the reference count of the variable container it points to is decreased by one. So, if we unset variable *$a* after running the above code, the reference count of the variable container that *$a* and element "1" point to gets decreased by one, from "2" to "1". This can be represented as:

Example #9 Unsetting *$a*

```
(refcount=1, is_ref=1)=array (
   0 => (refcount=1, is_ref=0)='one',
   1 => (refcount=1, is_ref=1)=...
)
```

Or graphically



Cleanup Problems¶

Although there is no longer a symbol in any scope pointing to this structure, it cannot be cleaned up because the array element "1" still points to this same array. Because there is no external symbol pointing to it, there is no way for a user to clean up this structure; thus you get a memory leak. Fortunately, PHP will clean up this data structure at the end of the request, but before then, this is taking up valuable space in memory. This situation happens often if you're implementing parsing algorithms or other things where you have a child point back at a "parent" element. The same situation can also happen with objects of course, where it actually is more likely to occur, as objects are always implicitly used by reference.

This might not be a problem if this only happens once or twice, but if there are thousands, or even millions, of these memory losses, this obviously starts being a problem. This is especially problematic in long running scripts, such as daemons where the request basically never ends, or in large sets of unit tests. The latter caused problems while running the unit tests for the Template component of the eZ Components library. In some cases, it would require over 2 GB of memory, which the test server didn't quite have.