COWS

Array Library

COWS Version 1.3

Sean Luke

March 20, 1994

The COWS Array library provides arrays of any number of dimensions up to 256, and of any length whatsoever.

Arrays and COWS

Because COWS has only one data type (the string), there are some odd things about the Array Library which are important to know before using it.

First off, arrays must be manually allocated and freed in COWS, much like object instances must be manually allocated and freed in Objective-C. This is because the *only* data that can be stored on the COWS stack is the string. So arrays cannot be created and placed on the stack, then bumped off by the interpreter. This means that you have to watch your allocation carefullyÐthere's no garbage collection in the Array Library, other than arrays getting wiped out when re-loading the library.

Second, arrays are "stored" as variables by placing a unique string in a variableÐa pointer to the array, as it were. This string is in the format "array X", where X is an integer that increases with the number of arrays currently allocated.

Lastly, arrays are always arrays of strings. But if you think about it, you can

make arrays of arrays as well, if you manually allocated an array for each cell of a grandfather array and place the pointer to it in the grandfather's cell.

The Library

The Array Library consists of two objects: the *COWSArrayLibrary* (the library itself), and the *COWSArrayNode,* (arrays stored in the library's hash table). Remember to include both in your project if you're building an application with COWS inside. But only connect the COWSArrayLibrary to your interpreter.

Creating an Array

You create arrays using the command:

(make-array *dimensions*+**)**

This returns a pointer to the array. *dimensions* is one or more values that define the dimensions of an array. For example, if you wanted to make a 4 x 3 x 2 array, you'd make it with

⇒ **(make-array 4 3 2)**

To store the array-pointer in a variable (important if you want to refer to the array later!), you'd do it like this (assuming the variable *array-var* had been allocated already):

⇒ **(set array-var (make-array 4 3 2))**

Setting a Value in an Array

You set a value in an array using the command:

(set-array *pointer value coordinates*+**)**

This returns *value.* *pointer* is the pointer to the array (see above), *coordinates* are the dimensional coordinates in the array for the cell you wish to set, and *value* is the value you want to set the cell to.

All arrays created with this library start counting at 1. So coordinates for the example above could range as [1±4], [1±3], [1±2]. This is different from C! In C, the range would be 0-3, 0-2, and 0-1.

Continuing the example above, let's say you wanted to set position 2,3,1 of *array-var* to the string "hello", position 4,1,2 to the number 5, and 1,2,2 to the value of the variable *val.* You'd set them like so:

⇒ **(set-array array-var "hello" 2 3 1)**
(set-array array-var 5 4 1 2)
(set-array array-var val 1 2 2)

Retrieving a Value in an Array

You get a value in an array using the command:

(array *pointer coordinates*+**)**

This returns the value of the cell of cordinate *coordinates* of the array pointed to by *pointer.* Once again, coordinates' ranges start at 1.

If you wanted to, say, print the value of the array stored in *array-var* at the coordinates 4,1,2, you'd write:

⇒ **(array array-var 4 1 2)**

Freeing an Array

You can free an array with the command:

(free-array *pointer***)**

This returns *t.* The array pointed to by *pointer* is freed, if it exists.

To free an array pointed to by *array-var,* you'd write:

⇒ **(free array-var)**

Determining an Array

If you're not sure if a pointer is pointing to a valid array, you can check with:

(array? *pointer***)**

This returns *t* if the array is valid, *f* if not.

To check an array pointed to by *array-var,* you'd write:

⇒ **(array? array-var)**

Array Function Names

In review, the function names are:

make-array Creates an array with dimensions *dimensions*.

Returns a pointer to the array.
(make-array *dimensions*+**)**

set-array Sets array *pointer* at coordinates *coordinates* to *value*.
Coordinates start at 1, not 0. Returns *value.*
(set-array *pointer value coordinates*+**)**

array Returns the value of array *pointer* at coordinates *coordinates*.
Coordinates start at 1, not 0.
(array *pointer coordinates*+**)**

free-array Frees array *pointer*. Returns *t.*
(free-array *pointer***)**

array? Returns *t* if *pointer* is a valid array, *f* otherwise.
(array? *pointer***)**