ARR v1.0

*by **digi_cs** Sep-Nov 2024* *cosmogen@gmail.com*

The ARR library provides basic functions for working with arrays in AWK.

The library includes 27 functions for array operations:

- 5 functions for joining array elements into a string:

ret::a	return array elements
ret::as	return array elements with a separator
ret::ad	return array elements
ret::ab	return elements of two arrays
ret::abs	return elements of two arrays with a separator

  These functions were created using elements of linear programming technologies, which is why they offer high performance and are designed for large data volumes and high numbers of merged indices.

  See the "**ARRAY JOINING**" section.

  See the "**PERFORMANCE**" section.

- 3 functions for declaring subarrays within arrays and removing indices from arrays:

def::ia	declare an empty subarray in array[index]
let::ia	declare a subarray in array[index] if it has not been declared as an array yet
del::ia	delete array[index]

- 2 special functions

arr::same	two DIFFERENT arrays?
arr::name	determinate name of the global array

- 2 functions for generating visualization of array contents

dump::a	array dump (currently not available)
dump::ab	dump of two arrays (currently not available)

- 15 functions for data transfers between arrays and their elements:

let::aa **def::aa** **mov::aa**	array1 < array2
let::iav **def::iav** **mov::iav**	array[index] < array/value
let::iai **def::iai** **mov::iai**	array[index1] < array[index2]
let::aia **def::aia** **mov::aia**	array1[index] < array2[index]
let::iaia **def::iaia** **mov::iaia**	array1[index1] < array2[index2]

The **let**, **def**, and **mov** functions differ in behavior when copying arrays into arrays:

mov-functions simply copy data from the source to the recipient: when copying an array into an already existing array, the data in it is preserved except for those that will be overwritten by the source indices during the copying process.

def-functions always clear the target: array or array[index]: exact copying.

let-functions copy only what is not present in the recipient: that is, only those source indices that are not in the recipient are copied.

USE

To use ithe library perform::

```
@include    "arr.lib"
```

The library features dynamic initialization, so the location where the library is included in the source code doesn't matter: resource initialization will happen automatically when needed.

The library has been tested for functionality and performance on GAWK 5.3.0

del::**ia**(*i*, *A*)

Delete *A*[*i*] (if exist)

Returns *i*.

let::**ia**(*i*, *A*)
def::**ia**(*i*, *A*)

Define *A*[*i*] as the subarray.

Returns *i*.

Specifuc:

def::ia always clear array *A*[*i*]

The following is examples of usage these functions:

```
example:
      patsplit( ..., A[ def::ia( "A", A ) ], …, A[ def::ia( "B", A ) ] )
```

patsplit will be targeted to arrays: A[“A”] and A[“B”]

let::**aa**(*D*, *S*)
def::**aa**(*D*, *S*)
mov::**aa**(*D*, *S*)

D[ ] < S[ ]

Copy all indexes from the source array *S* to destination array *D*.

if *D* and *S* is the same array then do nothing.

Returns *null*.

Specifuc:

def::aa pre-clear array *D*

let::aa indexes of *S* are will be copied in case if it's not exist in *D*.

examples:

```
# preparing the source array in SOURCE:

      SOURCE[ 1 ]             = "one"
      SOURCE[ 2 ][ "" ]       = "two"
      SOURCE[ 3 ]             = "three"

      print dump::a( SOURCE )

# create exact copy (snapshot) of the SOURCE in A

      A[ "" ] = "value A"                # this will be missed

      def::aa( A, SOURCE )

      print dump::a( A )


# add content of the SOURCE to B

      B[ "" ] = "value B"                # this will stay

      mov::aa( B, SOURCE )

      print dump::a( B )

# add content of the SOURCE that's not present in D

      C[ 1 ] = "value C"                 # this will stay unchanged

      let::aa( C, SOURCE )

      print dump::a( C )
```

outputs:

```
SOURCE[ 1 ] = one'
      [ 2 ][ "" ] = two'
      [ 3 ] = three'

A[ 1 ] = one'
 [ 2 ][ "" ] = two'
 [ 3 ] = three'

B[ "" ] = value B'
 [ 1 ] = one'
 [ 2 ][ "" ] = two'
 [ 3 ] = three'

C[ 1 ] = value C'
 [ 2 ][ "" ] = two'
 [ 3 ] = three'
```

let::iav(*i*, *A*, *V*)
def::**iav**(*i*, *A*, *V*)
mov::**iav**(*i*, *A*, *V*)

A[i] < V

Copy *V* to *A*[*i*]

V may be any of type (array or other).

Returns *i*

Specific:

let::iav, **mov::iav**	if *V* is untyped then do nothing
def::iav	if *V* is untyped then *A*[*i*] will be deleted
def::iav	*A*[*i*] will be always cleared before *V* copied

example:

```
# preparing data

      string            = "string"

      ARRAY[ "" ]       = "FROM ARRAY"

      A[ 1 ]            = "will be overwrited by mov::iav"

      A[ 2 ][ "" ]      = "will be unchanged by let::iav"

      A[ 3 ][ "" ]      = "will be overwrited by def::iav"
      A[ 3 ][ 1 ]       = "will be deleted by def::iav"

mov::iav( 1, A, string )

let::iav( 2, A, ARRAY )

def::iav( 3, A, ARRAY )

print dump::a( A )
```

outputs:

```
A[ 1 ] = string'
 [ 2 ][ "" ] = will be unchanged by let::iav'
 [ 3 ][ "" ] = FROM ARRAY'
```

let::iai(*d*, *A*, *s*)
def::iai(*d*, *A*, *s*)
mov::**iai**(*d*, *A*, *s*)

A[d] < A[s]

Copy element *s* in array *A* to *A*[*d*]

If *d* == *s* then do nothing.

Returns *d*

Specific:

let::iai, **mov::iai**	if *s* is not exist in *A* then do nothing
def::iai	if *s* is not exist in *A* then *A*[*d*] will be deleted
def::iai	if *d* is exist in *A* then it's will be pre-deleted

example:

```
print dump::a( A )
```

outputs:

let::aia(*D, i, S*)
def::aia(*D, i, S*)
mov::**aia**(*D, i, S*)

D[i] < S[i]

Copy *S*[*i*] to *D*[*i*]

If *D* and *S* are the same array then do nothing.

Returns *i*

Specific:

let::aia, **mov::aia**	if *i* is not exist in array *S* then do nothing
def::aia	if *i* is not exist in array *S* then *A*[*d*] will be deleted
def::aia	if *i* is exist in *D* then it's will be pre-deleted

example:

```
print dump::a( A )
```

outputs:

let::iaia(*d, D, s, S*)
def::iaia(*d, D, s, S*)
mov::**iaia**(*d, D, s, S*)

D[d] < ***S[s]***

Copy *S*[*s*] to *D*[*d*]

If *d* == *s* and *D* and *S* are the same array then do nothing.

Returns *d*

Specific:

let::iaia, **mov::iaia**	if *s* is not exist in array *S* then do nothing
def::iaia	if *s* is not exist in array *S* then *D*[*d*] will be deleted
def::iaia	if *d* is exist in *D* then it's will be pre-deleted

example:

```
      print dump::a( A )
```

outputs:

arr::**same**(*A*, *B*)

Returns true (1) in case if the given arrays *A* and *B* are the same array. Otherwise returns 0.

arr::**name**(*A*)

Returns the name of the given global array *A*.

If *A* is not global array then returns *null.*

dump::**a**(*A*, *name*)

Returns dump of the given array *A*

This function is currently not available

dump::**ab**(*A*, *B*)

Returns dump of the two given arrays *A* and *B*

This function is currently not available

ARRAY JOINING

This library includes the implementation of four operations related to combining array elements into a string:

```
RETA        array element merging

            A[a]  A[a+=x]  A[a+=x]

RETAS       joining array elements with separator

            A[a]  sep  A[a+=x]  sep  A[a+=x]

RETAB       pairwise merging of elements from two arrays

            A[a]B[b]  A[a+=x]B[b+=y]  A[a+=x]B[b+=y]

RETABS      pairwise merging of elements from two arrays with separator

            A[a]B[b]  sep  A[a+=x]B[ b+=y]  sep  A[a+=x]B[b+=y]
```

In the process of combining elements into a string, the data from the combined elements is read and copied into the resulting string. This process is called **data pass**.

Data pass refers to when some data is read and copied somewhere else. This process is what consumes performance during the joining of array elements.

When creating this library, the primary goal was to **minimize the number of data passes** for the same data. The number of required data passes directly d**epends on the number of indices being joined**.

Below is a table informing about the number of **data passes** for various amounts of combined indices:

indexes	**data pass**
1 .. 64	1
65 .. 64K	2
64K .. 4M	3
4M .. 256M	4
256M .. 16G	5

1 - 64 indices are returned in a single data pass—i.e., directly from the source arrays.

65 - 64K combined indices are returned in two data passes: the data is copied from the source arrays to a temporary array and then returned from it.

64K - 4M indices are returned in 3 data passes: the data is copied from the source arrays to a temporary array, the temporary array is compressed 64 times (AIR), and then the data is returned from it.

More than 4M and up to 256M combined indices are returned in 4 data passes: data is copied from the source arrays to a temporary array, the temporary array is compressed 64 times twice (AIR), and then the data is returned from it.

More than 256M and up to 16G (billions) of indices will require 5 data passes: data is copied from the source arrays to a temporary array, the temporary array is compressed 64 times three times (AIR), and then the data is returned from it.

The RETA operation has a double number of indices for each data pass—i.e., it returns up to 128 indices in one data pass and up to 128K in two data passes, and so on.

AIR

The AIR (Array Index Reduce) algorithm is based on the principle of combining an entire group (64) of incremental indices in an array into a single index within the same array. Then, the whole group of combined indices is removed from the array, and the operation is repeated until all the original indices in the array have been "compressed."

After the compression, a check is performed to determine the final number of indices in the array: does it exceed the threshold value (64)? If it does, the entire operation is repeated. If it does not, all the elements in the array are restored to a quantity equal to the threshold (64).

ret::**a**(*A, a, q, sa, sep*)
ret::**ad**(*A, sa, a, q, sep*)
ret::**as**(*A, sep, a, q, sa*)

All three functions are doing the same things and differs from each other only by parameters order.

Returns specified number of indexes of an array with separator string between each index:

```
return A[a]sep A[a+=sa]sep … A[a+=sa]
```

A is the source array

the first index in array is calculated by the following

```
first =  ! ( 0 in A )
```

first index is equals to 0 in case if index 0 is exist in array *A*
otherwise first index equals to 1

last index in an array is calculated by the following:

```
last = first + length( A ) - 1
```

a is the start index in array *A*

if *a* is omitted (== "") then it's will be calculated depending from *sa* parameter (see below):

if index modifier (*sa*) is equals to zero or positive number (>= 0) then start index *a* will be first index in array *A*

otherwise if index modifier (*sa*) is less than zero (< 0) then start index *a* will be last index in array *A*

sa is the index modifier for an array *A*

if *sa* is omitted (== "") then default value 1 will be used

q is the number of indexes to return

if *q* is omitted (== "") then it's will be calculated for covering rest of the elements of the array *A* starting from determinated start index *a* and using determinated index modifier *sa*

sep is the separator string that will be placed between the content of the each index joined

Below is the examples of usage functions:

example 1:

```
t = "ABCDE"

split( t, T, "" )            # splitting t for each character

t2 = ret::a( T )             # joining all elements of the array T

# t == t2 == "ABCDE"
```

example 2:

```
t = "ABCDE"

split( t, T, "" )            # splitting t for each character

t2 = ret::ad( T, -1 )        # joining all elements of the array T
                             # in reverse order
# t  == "ABCDE"
# t2 == "EDCBA"
```

example 3:

```
split( t, L, /\n/ )          # splitting multilined t for each line

t2 = ret::as( L, "\n" )      # joining all elements (lines of t) of
                             # the array L with separator: "\n" (eol)
# t == t2
```

example 4:

```
t = "ABCDE"

split( t, T, "" )            # splitting t for each character

t2 = ret::a( T, 3 )          # joining rest of the elements of the
                             # array T starting from index 3 using
                             # default index modifier (+1) with no
                             # separator
# t2 == "CDE"
```

example 5:

```
t = "ABCDE"

split( t, T, "" )            # splitting t for each character

t2 = ret::ad( T, -1, 3 )     # joining rest of the elements of the
                             # array T starting from index 3 and using
                             # index modifier -1 with no separator
# t2 == "CBA"
```

ret::**ab**(*A, B, a, q, b, sa, sb, sep*)
ret::**abs**(*A, B, sep, a, q, b, sa, sb*)

Both functions are doing the same things and differs from each other only by parameters order.

Returns specified number of index-pairs of the two arrays with separator string between each pair:

```
return A[a]B[b]sep A[a+=sa]B[b+=sb]sep … A[a+=sa]B[b+=sb]
```

A, B are the source arrays

the first index in array is calculated by the following

```
first =  ! ( 0 in A/B )
```

first index is equals to 0 in case if index 0 is exist in array *A*/*B* otherwise first index equals to 1

last index in an array is calculated by the following:

```
last = first + length( A/B ) - 1
```

a, *b* are the start indexes of the array *A*, *B* respectively

if *a*/*b* is omitted (== "") then it's will be calculated depending from *sa*/*sb* parameter (see below):

if index modifier is equals to zero or positive number (>= 0) then start index *a*/*b* will be first index in an array *A*/*B*

otherwise if index modifier is less than zero (< 0) then start index *a/b* will be last index in an array *A*/*B*

sa, *sb* are the index modifiers for array *A*, *B* respectively

if *sa*/*sb* is omitted (== "") then default value 1 will be used

if *sa* is specified but *sb* is omitted then *sb* will be equals to *sa*

q is the number of index-pairs to return

if *q* is omitted (== "") then it's will be calculated for covering rest of the elements of the array *A* starting from determinated start index *a* and using determinated index modifier *sa*

sep is the separator string that will be placed between the content of the each index-pair joined

example:

```
f = "somefunc"

if ( q = patsplit( t, T, /\x7F......\xFF/, D ) ) {

      for ( t = 1; t <= q; t++ )

            T[ t ] = @f( T[ t ] )

      t = ret::ab( T, D ) }
```

The provided example shows a fragment of **typical pointer processing in a string:** we split the source string into two arrays: pointers and the data between them. Then we call a certain function for each of the found pointers, and each time the result of this function is saved in the same array instead of the body of the found pointer. When we have "gone through" all the found pointers, we assemble the string back together by **combining the two arrays into a string using the `ret::ab` function**.

ARRAYS IN AWK

to define names (A,B,C,...) as the arrays use:

```
"" in A in B in C in ...
```

simpliest data accummulation in an array A is possible using

```
A[ length( A ) ] =
```
new added data

please note that A MUST to be defined as an array before

to pass undefined subarray A[i] use the def::ia/let::ia functions:

```
A[ def::ia( i , A ) ]
```

ret::ab() function is designed as the opposite to built-in split/patsplit functions called with the two target arrays specified

PERFORMANCE

PROSPECTS

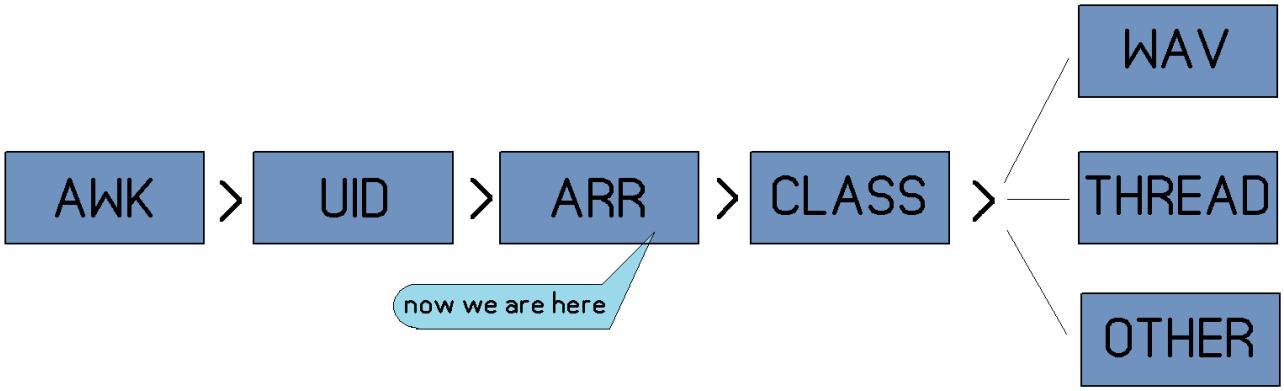
The next release will be the CLASS – the OOP implementation for AWK: the fundamentals of basic techniques and a description of OOP practices in AWK.

After class implementation will be published a number of classes will also be published:

class WAV – allowing to load .WAV file and split it to multiple files with specified beat-length and number of pieces. Potentially DSP features.

Class THREAD – allowing to execute/control/monitor/getdatafrom/kill external application in real-time using pstools

A hypothetical publication plan for AWK libraries and classes for the years 2024-2025:



AUTHOR

Class implementation in AWK is a key factor for the further comprehensive development of this language.

Thank you for your interest and attention!

Kind Regards
Denis Shirokov

RESPECT DUE

All awkers from all the world!

gawk Team