

NAME

array – declare a data array

Description

Unlike the “associative” arrays that have always been part of `spec`, data arrays (introduced in `spec` release 4) are more like those used in math and in programming. While associative arrays are indexed by arbitrary strings or numbers and can store either strings or numbers, data arrays are indexed by consecutive integers (starting from zero, as is the C convention) and hold a specific data type, such as short integer, float, double, etc.

Data arrays must be specifically declared and dimensioned using the `array` keyword (unlike associative arrays, which can come into existence when used in an expression). The arrays can have one or two dimensions. The `mca_get()` and `image_get()` functions can directly fill the arrays with data from one- or two-dimensional detectors.

Data arrays can be used in expressions containing the standard arithmetic operators to perform simultaneous operations on each element of the array. In addition, a subarray syntax provides a method for performing assignments, operations and functions on only portions of the array.

The functions `array_dump()`, `array_fit()`, `array_op()`, `array_plot()`, `array_pipe()` and `array_read()` handle special array operations. The functions `fmt_read()` and `fmt_write()` provide a means to transfer array data to and from binary-format data files. Also, the functions `mca_get()`, `mca_put()`, `image_get()` and `image_put()` accept array arguments, in addition to the original data-group syntax that uses group- and element-number arguments. (See the `data` help file for an explanation of data groups.)

The `print` command will print data arrays in a concise format on the screen, giving a count of repeated columns and rows, rather than printing each array element.

Another powerful feature of data arrays is that the array data can be placed in shared memory, making the data accessible to other processes, such as image-display or data-crunching programs. The shared arrays can both be read and written by the other processes. The implementation includes a number of special aids for making the processes work smoothly with `spec`.

Usage

Data arrays must be declared with the `array` key word. One- and two-dimensional arrays are declared as

```
[shared] [type] array var[cols]
[shared] [type] array var[rows][cols]
```

On platforms that support System V Interprocess Communication (IPC) calls, the `shared` keyword causes `spec` to place the array in shared memory (see below). The `type` keyword specifies the storage type of the array and may be one of `byte`, `ubyte`, `short`, `ushort`, `long`, `ulong`, `float`, `double` or `string`. An initial `u` denotes the “unsigned” version of the data type. The default data type is `double`.

The array name `var` is an ordinary `spec` variable name. Arrays are global by default, although they may also be declared `local` within statement blocks.

Unlike traditional `spec` associative arrays, which can store and be indexed by arbitrary strings or numbers, a data array is indexed by consecutive integers (starting from zero), and can hold only numbers, or in the case of `string` arrays, only strings.

Operations on these arrays can be performed on all elements of the array at once, or on one or more blocks of elements. Consider the following example:

```

array a[20]
a = 2
a[3] = 3
a[10:19] = 4
a[2,4,6,10:15] = 5

```

The first expression assigns the value 2 to all twenty elements of the array. The second expressions assigns 3 to one element. The third assign the value 4 to the tenth through last element. The final expression assigns the value 5 to the elements listed.

A negative number as an array index counts elements from the end of the array, with a [-1] referring to the last element of a.

As per the usual conventions, the first index is row and the second is column. Note, however, `spec` considers arrays declared with one dimension to be a single row. For example,

```
array a[20]
```

is a one-row, twenty-column array. Use

```
array a[20][1]
```

to declare a 20-row, one-column array.

Also note well, all operations between two arrays are defined as element-by-element operations, not matrix operations, which are currently unimplemented in `spec`. In the following example

```
array a[5][5], b[5][5], c[5][5]
```

```
c = a * b
```

`c[i][j]` is the product `a[i][j] * b[i][j]` for each `i` and `j`.

When two array operands have different dimensions the operations are performed on the elements that have dimensions in common. In the case

```
array a[5][5], b[5], c[5][5]
```

```
c = a * b
```

only the first row of `c` will have values assigned, since `b` only has one row. The remaining elements of `c` are unchanged by the assignment.

Portions of the array can be accessed using the subarray syntax, which uses colons and commas, as in the following examples.

```
array a[10][10]
```

```

a[1]                # second row of a
a[2:4][]           # rows 2 to 4
a[][2:]            # all rows, cols 2 to last
a[1,3,5,7,9][3:7] # odd rows and cols 3 to 7

```

The elements of an array can be accessed in reverse order, as in

```
a = x[-1:0]
```

which will assign to `a` the reversed elements of `x`. Note, though, that presently, an assignment such as `x = x[-1:0]` will not work properly, as `spec` will not make a temporary copy of the elements. However, `x = x[-1:0]+0` will work.

The functions `fabs()`, `int()`, `cos()`, `acos()`, `sin()`, `asin()`, `tan()`, `atan()`, `exp()`, `exp10()`, `log()`, `log10()`, `pow()` and `sqrt()` can all take arrays as an argument. The functions perform the operation on each element of the array argument and return the results in an array of the same dimension as the argument array.

The operations `<`, `<=`, `!=`, `==`, `>` and `>=` can be used with array arguments. The Boolean result (0 or 1) will be assigned to each element of an array returned as the result of the operation, based on the element-by-element comparison of the operands.

The bit-wise operators `~`, `|`, `&`, `>>` and `<<` can also be used with array operands.

String Arrays

Arrays of type `string` are identical to `byte` arrays in terms of storage requirements and behavior in most operations. However, when used as described below, the string arrays do behave differently.

If a row of a string array represents a number and is used in a conditional expression, then the value of the conditional expression will be the number. For example, the strings `"0.00"` or `"0x000"` will evaluate as zero or false in a conditional expression. In contrast, for number arrays, a conditional evaluates as zero only if every element of the array is zero.

Functions that take string arguments, such as `on()`, `length()`, `unix()`, etc., will allow a row of a string array to be used as an argument. Use of a number array is invalid and produces an error.

The `print` command will print string arrays as ASCII text, while byte arrays display each byte as a number.

In assignments to a row of a string array, the right hand side is copied to the byte elements of the string array as a string, even if the right hand side is a number. Any remaining elements of the string array row are set to zero. Thus, the results differ in the assignments below.

```
string array arr_string[20]
arr_string = 3.14159
print arr_string
3.14159
```

```
byte array arr_byte[20]
arr_byte = 3.14159
print arr_byte
{3 <20 repeats>}
```

In the first example, the string representation of the number is copied to the row of the string array, while in the second, each element of the array is assigned the (truncated) value of the number.

Row-wise and Column-wise Sense

For the functions `array_dump()`, `array_fit()`, `array_pipe()`, `array_plot()` and `array_read()` it matters whether each row or each column of a two-dimensional array corresponds to a data point. By default, `spec` takes the larger dimension to correspond to point number, and if both dimensions are the same, to use the rows as data points. The `"row_wise"` and `"col_wise"` arguments to `array_op()`, described below, can be used to force the sense of an array one way or the other, regardless of the array dimensions. If an array has row-wise sense, the contents of each row correspond to a data point, and one might then plot the contents of column two of each row versus column one, for example.

Shared Memory Arrays

When created with the `shared` keyword, the array data and a header structure are stored in shared memory. For each shared memory array, `spec` creates an immutable global variable named `SHMID_var` whose value is the shared memory ID associated with the shared memory segment and where `var` is the name of the array. This ID is used by other programs that wish to access the shared memory.

`spec` can connect to an existing shared memory array created by another process running on the same platform, perhaps created by another instance of `spec`. The syntax is

```
extern shared array [spec:[pid:]]arr
```

where the optional parameter `spec` is the name of the `spec` version that created the array, the optional parameter `pid` is the process ID of the version and `arr` is the name of the array. The first two arguments can be used in case more than one instance of the shared array exists. Examples include:

```
extern shared array data
extern shared array fourc:data
extern shared array fourc:1234:data
```

The shared array segments include a header that describes the array. Two features of the header that are primarily associated with shared arrays that can be accessed from `spec` user level are tags and frames. Shared arrays tags can be manipulated with the `array_op()` "tag" and "untag" options, as described in the next section.

Frame-size and latest-frame header elements allow a shared 2D array to be described as a series of 1D or 2D acquisitions (or frames). The frame size is the number of rows in a single frame. The latest frame is the most recently updated frame number. The latest frame value should allow an auxiliary program that maintains a live display to update the display efficiently. The frame values are also accessed via `array_op()`, below. Currently, the frame values are unused by `spec` in array operations, although specific hardware support may modify frames values. (Frames were added in `spec` release 5.08.06-1.)

The structure used for the shared memory data is given in the file `SPECD/include/spec_shm.h`. A C file containing an API for accessing the `spec` shared memory arrays is included in the `spec` distribution and is named `sps.c`.

Built-in Functions

`array_op("cmd", a, [args ...])` – Performs the following operations on the arguments as follows:

"fill" – Fills the array `a` with values. For a two-dimensional array,

```
array_op("fill", a, u, v)
```

produces

```
a[i][j] = u × i + v × j
```

With subarrays, `i` and `j` refer to the subarray index. Also, `i` and `j` always increase, even for reversed subarrays, so

```
array_op("fill", a[-1:0][-1:0], 1, 1)
```

fills `a` in reverse order.

"contract" – Returns a new array with dimensions contracted by a factor of `u` in rows and `v` columns. Elements of the new array are formed by averaging every `u` elements of each row with every `v` elements of each column. If there are left-over rows or columns, they are averaged also.

"min" or "gmin" – Returns the minimum value contained in the array.

- "max" or "gmax" – Returns the maximum value contained in the array.
- "i_at_min" or "i_at_gmin" – Returns the *index* number of the minimum value of the array. For a two-dimensional array dimensioned as $D[N][M]$, the index number of element $D[i][j]$ is $(i * M) + j$.
- "i_at_max" or "i_at_gmax" – Returns the *index* number of the maximum value of the array. See above.
- "row_at_min" or "rmin" – Returns the row number containing the minimum value of the array.
- "row_at_max" or "rmax" – Returns the row number containing the maximum value of the array.
- "col_at_min" or "cmin" – Returns the column number containing the minimum value of the array.
- "col_at_max" or "cmax" – Returns the column number containing the maximum value of the array.
- "i_<=_value" – Returns the *index* number of the nearest element of the array with a value at or less than u . For a two-dimensional array dimensioned as $D[N][M]$, the index number of element $D[i][j]$ is $(i * M) + j$.
- "i_>=_value" – Returns the *index* number of the nearest element of the array with a value at or greater than u , starting from the last element. For a two-dimensional array dimensioned as $D[N][M]$, the index number of element $D[i][j]$ is $(i * M) + j$.
- "fwhm" – Requires two array arguments, each representing a single row or single column. Returns the full-width in the first array at half the maximum value of the second array.
- "cfwhm" – Requires two array arguments, each representing a single row or single column. Returns the center of the full-width in the first array at half the maximum value of the second array.
- "uhmx" – Requires two array arguments, each representing a single row or single column. Returns the value in the first array corresponding to half the maximum value in the second array and at a higher index.
- "lhmx" – Requires two array arguments, each representing a single row or single column. Returns the value in the first array corresponding to half the maximum value in the second array and at a lower index.
- "com" – Requires two array arguments, each representing a single row or single column. Returns the center of mass in the first array with respect to the second array. The value is the sum of the products of each element of the first array and the corresponding element of the second array, divided by the number of points.
- "x_at_min" – Requires two array arguments, each representing a single row or single column. Returns the element in the first array that has the corresponds to the minimum value in the second array.
- "x_at_max" – Requires two array arguments, each representing a single row or single column. Returns the element in the first array that has the corresponds to the maximum value in the second array.
- "sum" or "gsum" – Returns the sum of the elements of the array. If there is a third argument greater than zero, the array is considered as a sequence of frames, with the third argument the number of rows in each frame. The return value is a new array with that number of rows and the same number of columns as the original array. Each element of the returned array is the sum of the corresponding elements of each frame. For example, if the original array is dimensioned as $data[N][M]$, the return value for

```
a = array_op("sum", data, R)
```

is a new array of dimension $a[N/R][M]$, where each element $a[i][j]$ is the sum of k from 0 to $R - 1$ of $data[i + k * N / R][j]$.

- "sumsq" – Returns the sum of the squares of the elements of the array. If there is a third argument and it is greater than zero, the interpretation is the same as above for "sum", except the elements in the returned array are sums of squares of the elements in the original array.
- "transpose" – Returns a new array of the same type with the rows and columns switched.
- "updated?" – Returns nonzero if the data in the array has been accessed for writing since the last check, otherwise returns zero.
- "rows" – Returns the number of rows in the array.
- "cols" – Returns the number of columns in the array.
- "row_wise" – With a nonzero third argument, forces the `array_dump()`, `array_fit()`, `array_pipe()`, `array_plot()` and `array_read()` functions to treat the array as row-wise, meaning each row corresponds to a data point. With only two arguments, returns nonzero if the array is already set to row-wise mode.
- "col_wise" – As above, but sets or indicates the column-wise sense of the array.
- "sort" – Returns an ascending sort of the array.
- "swap" – Swaps the bytes of the named array. The command can change big-endian short- or long-integer data to little-endian and vice versa. For most built-in data collection, `spec` automatically swaps bytes as appropriate, but this function is available for other cases that may come up.
- "frame_size" – The number of rows in a frame. The frame size is part of the shared array header and may be useful to auxiliary programs, although the value is maintained for non-shared arrays. Note, setting the frame size to zero will clear the "frames" tag. Setting the frame size to a non-zero value will set the "frames" tag.
- "latest_frame" – The most recently updated frame. The latest frame is part of the shared array header and may be useful to auxiliary programs, although the value is maintained for non-shared arrays.
- "tag" – Shared arrays can be tagged with a type that will be available to other processes accessing the array. Usage is `array_op("tag", arr, arg)` where `arr` is the array and `arg` is "mca", "image", "frames", "scan" or "info".
- "untag" – Removes tag information.

`array_fit(pars, a [, b, ...])` – Performs a linear fit of the data in the array `a`. The fitted parameters are returned in the array `pars`. The function returns the *chi-squared* value of the fit, if the fit was successful. A `-1` is returned if the covariance matrix is singular. The fit algorithm is along the same lines as the `lfit()` routine in *Numerical Recipes* (W.H. Press, et al., Cambridge University Press, 1986, page 512).

`array_dump([file,] a, [, b, ...] [, options, ...])` – Efficiently writes the data in the array `a` and optionally arrays `b, ..., etc`. If the initial optional `file` argument is given, the output is just to the named file or device. Otherwise, output is to all "on" output devices. The additional `options` arguments are strings that each start with a percent sign as follows:

An optional format argument can specify a `printf()`-style format for the values. The default format is `"%.9g"`, which prints nine digits of precision using fixed point or exponential format, choosing whichever is more appropriate to the value's magnitude. Recognized alternate format characters are `e` or `E` (exponential), `f` (fixed point), `g` or `G` (fixed or exponential based on magnitude), `d` (decimal integer), `u` (unsigned integer), `o` (octal integer), `x` or `X` (hexadecimal integer). (The last two as of release 5.08.01-1, and

d, u and o as of release 5.08.03-6.) All formats accept standard options such as precision and field width. For example, "%15.8f" uses fixed-point format with eight digits after the decimal point and a fifteen-character-wide field. For the integer formats, double values will be converted to integers. Also, initial characters can be included in the format string, for example, "0x%08x" is valid.

The option "%D=c", specifies an alternate delimiter character *c* to replace the default space character delimiter that is placed between each element in a row of output. For example, one might use a comma, a colon or the tab character with "%D=", "%D=" or "%D=\t", respectively. Use "%D=" for no delimiter.

Also, by default, the output is one data row per line. Thus, for one-dimensional row-wise arrays, all elements will be printed on one line, while one-dimensional column-wise array will have just one data element per line. For two-dimensional arrays, each line will contain one row of data. The number of elements per line can be controlled with the option "%#[C|W]". For one-dimensional arrays, the number # is the number of elements to print per line. For two-dimensional arrays, # is the number of rows to print per line. If an optional W is added, the number becomes the number of elements to print per line, which can split two-dimensional arrays at different points in the rows. If an optional C is added to the option string, a backslash will be added to each line where a row is split. (The C-PLOT *scans.4* user function can properly interpret such "continued" lines for one-dimensional MCA-type array data.)

Finally, the various options can be combined in a single string. For example,

```
a = array_dump(data, "%15.4f", "%D=: ", "%8W")
```

and

```
a = array_dump(data, "%15.4f%D=:%8W")
```

work the same.

`array_plot(a [, b, ...])` – Plots the data in the array *a*. Depending on whether *a* is a row-wise or column-wise array, the first column or first row elements are used for *x*. Subsequent elements (up to a maximum of 64) are plotted along the *y* axis. If preceded by a call of `plot_cntl("addpoint")` and the ranges have not changed, only the last point in the array is drawn. If preceded by a call of `plot_cntl("addline")` the current plot will not be erased, and the plot ranges will not be changed. The plotting area is not automatically erased by a call of `data_plot()`—use `plot_cntl("erase")` for that. The axis ranges are set using the `plot_range()` function. See `plot_cntl()` for other options that affect drawing the plot.

`array_read(file_name, a)` – Reads data from the ASCII file *file_name*, and stuffs the data into the array *a*. For a row-wise array, the values on each line of the file are assigned into successive columns for each row of the array. If there are more items on a line in the file than columns in the array, or if there are more points in the file than rows in the array, the extra values are ignored. For a column-wise array, each row of the data file is assigned to successive columns of the array. Lines beginning with the # character are ignored. Returns -1 if the file can't be opened, otherwise returns the number of points read. At present, the maximum input line length is 2,048 characters.

`fmt_read(file, format, array [, header [, flags]])` – Reads data from *file* using the indicated *format* into *array*, possibly assigning elements of the associative array or string *header*. Values for *flags* depend on the implementation for the particular file format. Contact CSS directly for more information on the `fmt_read()` function.

`fmt_write(file, format, array [, header [, flags]])` – Saves data to *file* using the indicated *format* from *array*, possibly also assigning elements of the associative array or string *header*. Values for *flags* depend on the implementation for the particular file format. Contact CSS directly for more information on the `fmt_write()` function.

`fmt_close(file, format)` – Calls the file close routine associated with *format* for *file*.