

# ARRAY

**View online:** <https://www.construct.net/en/make-games/manuals/construct-3/plugin-reference/array>

---

The **Array** object stores lists of values (numbers or text). It is analogous to arrays in traditional programming languages.

## Scripting

When using JavaScript or TypeScript coding, the features of this object can be accessed via the **IArrayInstance** script interface. (JavaScript and TypeScript have built-in support for arrays, but this allows interacting with an array used in event sheets.)

## About Arrays

Array supports up to three dimensions. For example, a simple list of ten values would be a  $10 \times 1 \times 1$  array. Note that you should not set a size of 0 on any of the dimensions else the *entire* array becomes empty; it is correct to have a size of 1 on unused dimensions.

Each element of an array can store a number or some text. The number or text in an element can be changed with the *Set* actions, and accessed with the *At* expression. For example, a  $10 \times 10 \times 1$  array is analogous to a 2D grid with a total of 100 values. A number could be stored at the position (3, 7) with the action *Set at XY*, and accessed with `Array.At(3, 7)`. Note like the rest of Construct indices are zero-based, so the first element is at 0. In this example, `Array.At(0, 0)` would return the first number in the grid.

Array can store either text or a number in any of the elements. Numbers and text can also be mixed within an array.

Arrays do not automatically resize. If you access a value outside the array bounds, it returns the number 0. If you set a value outside the array bounds, it will have no effect.

## Designing arrays

You can use Construct's **Array Editor** Paid plans only to set the initial contents of an array. You can create a new array data file as a **project file** from the **Project Bar**. At runtime you can load the project file with the **AJAX** object and use the Array's *Load* action to read the data file from the AJAX's *LastData* expression.

## Manipulating arrays

A one-dimensional array, sized  $N \times 1 \times 1$ , serves as a simple list of N values. The actions in the *Manipulation* category (e.g. *Push*, *Pop*) allow one-dimensional arrays to be used like other data structures. (These actions work with multidimensional arrays, but are intended for the one-dimensional case.)

For example, the following scheme implements a queue (first in first out, or 'FIFO'):

- Add new items with **Push front**
- Retrieve the next value with `Array.Back`
- Remove the retrieved value with **Pop back**

The following scheme implements a stack (last in first out, or 'LIFO'):

- Add new items with **Push back**
- Retrieve the next value with `Array.Back`
- Remove the retrieved value with **Pop back**

## Array properties

---

### **Width (X dimension)**

### **Height (Y dimension)**

### **Depth (Z dimension)**

The size of the array. If you want a one-dimensional array (i.e. a list of values), use A x 1 x 1.

If you want a two-dimensional array (i.e. a grid of values) use A x B x 1.

---

### **Elements Read-only**

This property indicates the total number of elements in the array. If it is 0, the array is completely empty and is unable to store any data. A common mistake is to set the Y or Z axis sizes to 0 which causes the entire array to be empty; this property helps you identify this mistake. Also using a huge array can cause very high memory use, and the element count helps you identify this case as well.

## Array conditions

---

### **Compare at X**

### **Compare at XY**

### **Compare at XYZ**

Compare a value at a position in the array. Indices are zero-based. All values outside the array return the number 0. If *Compare at X* is used, the Y and Z indices are 0. If *Compare at XY* is used, the Z index is 0.

---

### **Compare size**

Compare the size of one of the array dimensions, which is the number of elements on that axis.

## For each element

A repeating condition that runs once for each element in the array. This therefore runs *width*  $\times$  *height*  $\times$  *depth* times.

## Compare current value

Only valid in a *For each element* loop, either as a following condition or in a [sub-event](#). This compares the current value being iterated in the loop.

## Contains value

Searches the entire array to check if any of the elements contains the given value. For example, you can use this to test if the string "sword" is stored anywhere in the array.

## Is empty

Test if the array is empty. The array is empty when the total number of elements is zero, calculated as *width*  $\times$  *height*  $\times$  *depth*. Therefore the array is empty when any axis has a size of zero. This can be useful when using Array as a data structure (e.g. when pushing and popping values).

# Array actions

## Clear

Set every element in the array to the given value, which by default is the number 0.

## Set at X

## Set at XY

## Set at XYZ

Write a value at a position in the array. Indices are zero-based. Writing to values outside the array has no effect. If Set at X is used, the Y and Z indices are 0. If Set at XY is used, the Z index is 0.

## Set size

Set the dimensions of the array. Values are preserved, but if the new array is smaller it is truncated. If the new array is larger, new elements are set to store the number 0. If any of the dimensions are 0 the entire array is empty, so usually all the dimensions are at least 1.

## Download

Invokes a browser download of a file containing the Array's contents in JSON format.

## Load

Load the contents of the array from a string in JSON format. This can be retrieved from either the *Download* action, the *AsJSON* expression, or the *AJAX* object loading a project file.

*Note this does not allow loading arbitrary JSON data - the data must be in a special format for Construct. If you want to load data from a project file, create it using the New - Array menu option in the Project Bar.*

## Push

Add a new value either to the beginning (front) or end (back) of an axis. Since the Array is a 3D cube of values, technically this inserts a new 2D plane of elements all with the given value. However in 1D arrays this adds a single element, and in 2D arrays it inserts a new row of elements.

## Pop

Delete the value at either the beginning (front) or end (back) of an axis. Since the Array is a 3D cube of values, technically this removes a 2D plane of elements. However in 1D arrays this removes a single element, and in 2D arrays it removes a whole row of elements.

## Insert

Insert a new value at a specific index on an axis. Since the Array is a 3D cube of values, technically this inserts a new 2D plane of elements all with the given value. However in 1D arrays this adds a single element, and in 2D arrays it inserts a new row of elements.

## Delete

Delete the value at a specific index on an axis. Since the Array is a 3D cube of values, technically this removes a 2D plane of elements. However in 1D arrays this removes a single element, and in 2D arrays it removes a whole row of elements.

## Reverse

Reverse the order of elements on an axis. Note that in multidimensional arrays this only reverses one axis. For example reversing the X axis in a 2D array will reverse the order of the columns while preserving the contents of each column.

## Shuffle

Sort elements along a given axis into a random order. When using 2D or 3D arrays, then one-dimensional arrays are independently shuffled depending on the chosen axis. For example when choosing the X axis, every row is independently shuffled; when choosing the Y axis, every column is independently shuffled.

## Sort

Sorts the order of elements on an axis in ascending order. There are a variety of ways to sort multidimensional arrays. The options are:

- **X axis (by column):** sort the X axis by the first row of elements, moving entire columns when rearranging the ordering. Where elements match in the first row, it will refer to further elements down the Y axis as tie breakers.
- **X axis (separately):** sort the X axis independently in every row.
- **Y axis (by row):** sort the Y axis by the first column of elements, moving entire rows when rearranging the ordering. Where elements match in the first column, it will refer to further elements down the X axis as tie breakers.
- **Y axis (separately):** sort the Y axis independently in every column.
- **Z axis:** sort the Z axis independently in every 2D (X/Y) position in the array.

## Split string

Sets the array to a one-dimensional list of items based on splitting a string by a certain character. For example splitting the string `"1,2,3"` with the separator `,` and type `Auto` will set the array size to  $3 \times 1 \times 1$  with the numbers 1, 2 and 3. The `Type` parameter determines whether values are read as strings or numbers. The default mode `Auto` will set values as numbers if the token looks like a number, and a string if not. Setting the type to `String` or `Number` will ensure all values are consistently set as the given data type. The array can be converted back to a string with the `JoinString` expression.

*Splitting string can work for simple cases but has limitations. For example it's not possible for the separator to appear inside values, and it is not possible to explicitly specify the data type of values. To handle more complex cases, use a more robust data format like [JSON](#).*

## Array expressions

### At(x)

### At(x, y)

### At(x, y, z)

Retrieve a value at a position in the array. Indices are zero-based. Reading values outside the array returns the number 0. If the Y or Z indices are not provided then 0 is used.

## **CurX**

## **CurY**

## **CurZ**

The current zero-based index for each dimension in a *For each element* loop.

---

## **CurValue**

The current value in a *For each element* loop. This is a shortcut for `Array.At(Array.CurX, Array.CurY, Array.CurZ)`.

---

## **Width**

## **Height**

## **Depth**

Return the size of each of the array's dimensions.

---

## **Front**

Shortcut to access the first value in the array, which is the same as `At(0, 0, 0)`.

---

## **Back**

Shortcut to access the last value on the X axis, which is the same as `At(Self.Width - 1, 0, 0)`.

---

## **IndexOf**

## **LastIndexOf**

Searches the array X axis for a given value and returns the index it is found at, or -1 if not found. *IndexOf* finds the first matching element, and *LastIndexOf* finds the last matching element.

---

## **AsJSON**

Return the contents of the array as a string in JSON format. This can later be loaded in to the array with the *Load* action.

---

## **JoinString(separator)**

Convert a one-dimensional array to a string by converting every value to a string and joining them together with a separator. For example if an array is sized 3 x 1 x 1 and stores the values 1, 2 and 3, then the expression `Array.JoinString(",")` will return the string `1,2,3`. This is effectively the reverse of the *Split string* action.