# Arrays



## Introduction

Arrays can be used to store multiple values in one data structure. Think of them as a better way to handle cases where you would otherwise need a set of variables named `price00`, `price01` and `price02`. Arrays are an advanced feature used for scripts requiring intricate data-handling. If you are a beginning Pine Script™ programmer, we recommend you become familiar with other, more accessible Pine Script™ features before you tackle arrays.

Pine Script™ arrays are one-dimensional. All elements of an array are of the same type, which can be “int”, “float”, “bool”, “color”, “string”, “line”, “label”, “box” or “table”, always of “series” form. Arrays are referenced using an array ID similar to line or label IDs. Pine Script™ does not use an indexing operator to reference individual array elements; instead, functions like array.get() and array.set() are used to read and write values of array elements. Array values can be used in all Pine Script™ expressions and functions where a value of “series” form is allowed.

Elements within an array are referred to using an *index*, which starts at 0 and extends to the number or elements in the array, minus one. Arrays in Pine Script™ can be sized dynamically, so the number of elements in the array can be modified within one iteration of the script on a bar, and vary across bars. Multiple arrays can be used in the same script. The size of arrays is limited to 100,000.

> **Note**
> We will use *beginning* of an array to designate index 0, and *end* of an array to designate the array’s element with the highest index value. We will also extend the meaning of *array* to include array IDs, for the sake of brevity.

## Declaring arrays

The following syntax can be used to declare arrays:

```
<type>[] <identifier> = <expression>
var <type>[] <identifier> = <expression>
array<type> <identifier> = <expression>
var array<type> <identifier> = <expression>
```

The `[]` modifier (not to be confused with the [] history-referencing operator) is appended to the type name when declaring arrays. However, since type-specific functions are always used to create arrays, the `<type>[]` part of the declaration is redundant, except if you initialize an array variable to `na`. Explicitly declaring the array type helps state our intention to readers more clearly.

In the following example, we declare an array variable named `prices` and initialize it with `na`. Consequently, its type must be specified. The variable will be used to designate an array containing “float” values, but no array has been created by this declaration yet. For the moment, the array variable contains no valid array ID, its value being `na`:

```
float[] prices = na
```

We could also write the above example in this alternate form:

```
array<float> prices = na
```

When declaring an array and the `<expression>` is not `na`, one of the following functions must be used: `array.new_<type>(size, initial_value)`, array.from(), or array.copy(). For the `array.new_<type>(size, initial_value)` functions, the arguments of the `size` and `initial_value` parameters can be “series” to allow dynamic sizing and initialization of array elements. The following example creates an array containing zero “float” elements, and this time, the array ID returned by the array.new_float() function call is assigned to `prices`:

```
prices = array.new_float(0)
```

Similar array creation functions exist for the other types of array elements: array.new_int(), array.new_bool(), array.new_color(), array.new_string(), array.new_line(), array.new_linefill(), array.new_label(), array.new_box() and array.new_table().

When declaring an array using one of the array creation functions, you can initialize all elements using the `initial_value` parameter. When no argument is supplied for `initial_value`, the array elements are initialized to `na`. The following declaration creates an array ID named `prices`. The array is created with two elements, each initialized with the value of the `close` built-in variable on that bar:

```
prices = array.new_float(2, close)
```

You can also use array.from() to create an array and initialize it with different values at the same time. array.from() infers the array's size and the type of its elements, which must be consistent, from the arguments supplied to the function when calling it. Similarly to ``array.new_*()` functions, it accepts "series" arguments.

All three of the lines of code below will create identical "bool" arrays with the same two elements:

```
statesArray = array.from(close > open, high != close)
bool[] statesArray = array.from(close > open, high != close)
array<bool> statesArray = array.from(close > open, high != close)
```

## Using the `var` keyword

The var keyword can be used when declaring arrays. It works just as it does for other variables; it causes the declaration to only be executed on the first iteration of the script on the dataset's bar at bar_index zero. Because the array is never re-initialized on subsequent bars, its value will persist across bars, as the script iterates on them.

When an array declaration is done using var and a new value is pushed at the end of the array on each bar, the array will grow by one on each bar and be of size `bar_index + 1` plus one (bar_index starts at zero) by the time the script executes on the last bar, as this code will do:

```
//@version=5
indicator("Using `var`")
var a = array.new_float(0)
array.push(a, close)
if barstate.islast
    label.new(bar_index, 0, "Array size: " + str.tostring(array.size(a)) + "\nbar_index
```

The same code without the var keyword would re-declare the array on each bar. After execution of the array.push() call, the array would thus be of size one on all the dataset's bars.

This initializes an array of constant lengths which will not change during the script's execution, so we only declare it on the first bar:

```
var int[] lengths = array.from(2, 12, 20, 50, 100, 200)
```

## Reading and writing array values

Values can be written to existing individual array elements using array.set(id, index, value), and read using array.get(id, index). As is the case whenever an array index is used in your code, it is imperative that the index never be greater than the array's size, minus one (because array indices start at zero). You can obtain the size of an array by using the array.size(id) function.

The following example uses array.set() to initialize an array of colors to instances of one base color using different transparency levels. It then fetches the proper array element to use it in a bgcolor() call:


```
//@version=5
indicator("Distance from high", "", true)
lookbackInput = input.int(100)
FILL_COLOR = color.green
// Declare array and set its values on the first bar only.
var fillColors = array.new_color(5)
if barstate.isfirst
    // Initialize the array elements with progressively lighter shades of the fill colo
    array.set(fillColors, 0, color.new(FILL_COLOR, 70))
    array.set(fillColors, 1, color.new(FILL_COLOR, 75))
    array.set(fillColors, 2, color.new(FILL_COLOR, 80))
    array.set(fillColors, 3, color.new(FILL_COLOR, 85))
    array.set(fillColors, 4, color.new(FILL_COLOR, 90))

// Find the offset to highest high. Change its sign because the function returns a nega
lastHiBar = - ta.highestbars(high, lookbackInput)
// Convert the offset to an array index, capping it to 4 to avoid a runtime error.
// The index used by `array.get()` will be the equivalent of `floor(fillNo)`.
fillNo = math.min(lastHiBar / (lookbackInput / 5), 4)
// Set background to a progressively lighter fill with increasing distance from locatio
bgcolor(array.get(fillColors, fillNo))
// Plot key values to the Data Window for debugging.
plotchar(lastHiBar, "lastHiBar", "", location.top, size = size.tiny)
plotchar(fillNo, "fillNo", "", location.top, size = size.tiny)
```

Another technique that can be used to initialize the elements in an array is to declare the array with size zero, and then populate it using array.push() to append **new** elements to the end of the array, increasing the size of the array by one at each call. The following code is functionally identical to the initialization section from the preceding script:

```
// Declare array and set its values on the first bar only.
var fillColors = array.new_color(0)
if barstate.isfirst
    // Initialize the array elements with progressively lighter shades of the fill colo
    array.push(fillColors, color.new(FILL_COLOR, 70))
    array.push(fillColors, color.new(FILL_COLOR, 75))
    array.push(fillColors, color.new(FILL_COLOR, 80))
    array.push(fillColors, color.new(FILL_COLOR, 85))
    array.push(fillColors, color.new(FILL_COLOR, 90))
```

Finally, we could use array.from():

```
//@version=5
indicator("Using `var`")
FILL_COLOR = color.green
var color[] fillColors =
  array.from(
   color.new(FILL_COLOR, 70),
   color.new(FILL_COLOR, 75),
   color.new(FILL_COLOR, 80),
   color.new(FILL_COLOR, 85),
   color.new(FILL_COLOR, 90))
// Cycle background through the array's colors.
bgcolor(array.get(fillColors, bar_index % array.size(fillColors)))
```

The array.fill(id, value, index_from, index_to) function can be used to fill contiguous sets of array elements with a value. Used without the last two optional parameters, the function fills the whole array, so:

```
a = array.new_float(10, close)
```

and:

```
a = array.new_float(10)
array.fill(a, close)
```

are equivalent, but:

```
a = array.new_float(10)
array.fill(a, close, 1, 3)
```

only fills the second and third elements (at index 1 and 2) of the array with `close`. Note how array.fill()'s last parameter, `index_to`, needs to be one greater than the last index to be filled. The remaining elements will hold the `na` value, as no intialization value was provided when the array was declared.

## Looping through array elements

When looping through array elements when the array's size is unknown, you can use:

```
//@version=5
indicator("Protected `for` loop", overlay = true)
array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close)

string labelText = ""
for i = 0 to (array.size(a) == 0 ? na : array.size(a) - 1)
    labelText += str.tostring(array.get(a, i)) + "\n"

label.new(bar_index, high, text = labelText)
```

**Note that:**

- We use the request.security_lower_tf() function which returns an array of close prices at the `1 minute` timeframe.
- This code example will throw an error if you use it on a chart timeframe smaller than `1 minute`.
- for loops do not execute if the `to` expression is na. Note that the `to` value is only evaluated once upon

entry.

A much more recommended method to loop through array elements when the array's size is unknown is to use a `for...in` loop. This method is a variation of the traditional for loop that dynamically adjusts the number of iterations based on the array's size. Here is an example of how you can write the code example from above using this method:

```
//@version=5
indicator("`for...in` loop", overlay = true)
array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close)

string labelText = ""
for price in a
    labelText += str.tostring(price) + "\n"

label.new(bar_index, high, text = labelText)
```

A `while` loop statement can also be used:

```
//@version=5
indicator("`while` loop", overlay = true)
array<float> a = request.security_lower_tf(syminfo.tickerid, "1", close)

string labelText = ""
int i = 0
while i < array.size(a)
    labelText += str.tostring(array.get(a, i)) + "\n"
    i += 1

label.new(bar_index, high, text = labelText)
```

## Scope

Arrays can be declared in a script's global scope, as well as in the local scope of a function or an `if` branch. One major distinction between Pine Script™ arrays and variables declared in the global scope, is that global arrays can be modified from within the local scope of a function. This new capability can be used to implement global variables that can be both read and set from within any function in the script. We use it here to calculate progressively lower or higher levels:

```
//@version=5
indicator("Bands", "", true)
factorInput = 1 + (input.float(-2., "Step %") / 100)
// Use the lowest average OHLC in last 50 bars from 10 bars back as the our base level.
level = array.new_float(1, ta.lowest(ohlc4, 50)[10])

nextLevel(val) =>
    newLevel = array.get(level, 0) * val
    // Write new level to the global array so it can be used as the base in the next ca
    array.set(level, 0, newLevel)
    newLevel

plot(nextLevel(1))
plot(nextLevel(factorInput))
plot(nextLevel(factorInput))
plot(nextLevel(factorInput))
```

## History referencing

Past instances of array IDs or elements cannot be referenced directly using Pine Script™'s `[]` history-referencing operator. One **cannot** write: `array.get(a[1], 0)` to fetch the value of the array's first element on the previous bar.

In Pine Script™, however, each call to a function leaves behind a series trail of function results on previous bars. This series can in turn be used when working with arrays. One can thus write: `ma = ta.sma(array.get(a, 0), 20)` to calculate the simple moving average of the value returned by the `array.get(a, 0)` call on the last 20 bars.

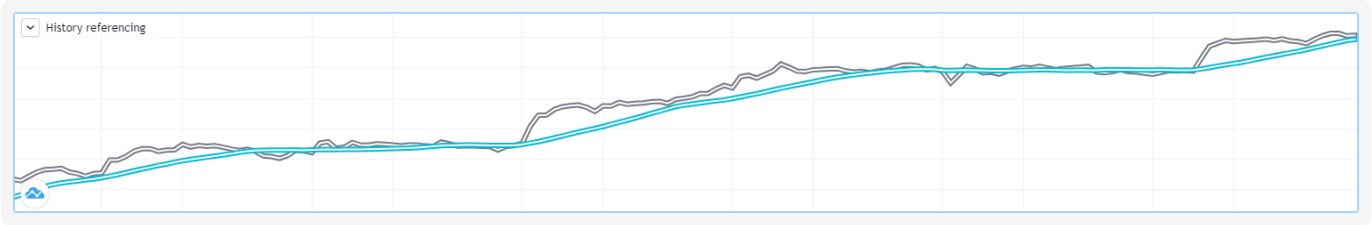
To illustrate this, let's first see how we can fetch the previous bar's `close` value in two, equivalent ways. For `previousClose1` we use the result of the `array.get(a, 0)` function call on the previous bar. Since on the previous bar the array's only element was initialized to that bar's `close` (as it is on every bar), referring to `array.get(a, 0)[1]` returns that bar's `close`, i.e., the value of the `array.get(a, 0)` call on the previous bar.

For `previousClose2` we use the history-referencing operator to fetch the previous bar's `close` in normal Pine Script™ fashion:

```
//@version=5
indicator("History referencing")
// Re-declare the array on each bar.
a = array.new_float(1)
// Set the value of its only element to `close`.
array.set(a, 0, close)

previousClose1 = array.get(a, 0)[1]
previousClose2 = close[1]
plot(previousClose1, "previousClose1", color.gray, 6)
plot(previousClose2, "previousClose2", color.white, 2)
```

In the following example we add two, equivalent calculations of a moving average to our previous code example. For `ma1` we use ta.sma() on the series of values returned by the `array.get(a, 0)` function call on each bar. Since at this point in the script the call returns the current bar's `close`, that is the value used for the average's calculation. We evaluate `ma2` using the usual way we would calculate a simple average in Pine Script™:




```
//@version=5
indicator("History referencing")
a = array.new_float(1)
array.set(a, 0, close)
previousClose1 = array.get(a, 0)[1]
previousClose2 = close[1]
plot(previousClose1, "previousClose1", color.gray, 6)
plot(previousClose2, "previousClose2", color.white, 2)

ma1 = ta.sma(array.get(a, 0), 20)
ma2 = ta.sma(close, 20)
plot(ma1, "MA 1", color.aqua, 6)
plot(ma2, "MA 2", color.white, 2)

// Last set having no impact.
array.set(a, 0, 10.0)
```

Notice the last line of this script. It illustrates how even if we set the value of the array's element to `10.0` at the end of the script, resulting in the final value for the element being committed as `10.0` on the bar's last execution of the script, the earlier call to `array.get(a, 0)` nonetheless returned the `close` value because that was the value of the array element at that point in the script. The series value of the function call will thus be each bar's `close` value.

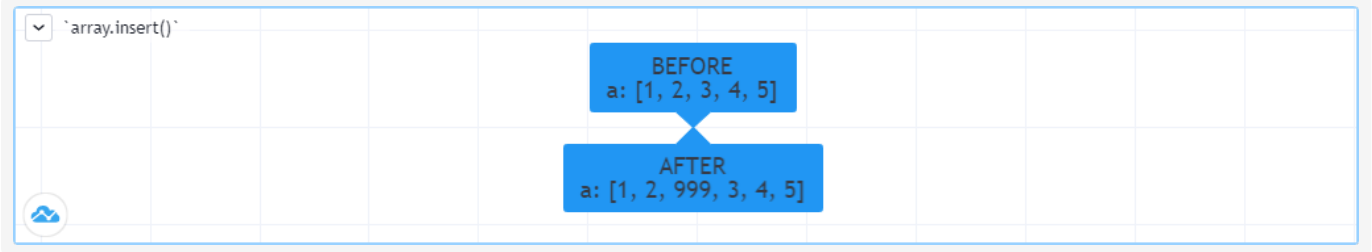
# Inserting and removing array elements

## Inserting

Three functions can be used to insert new elements in an array.

array.unshift() inserts a new element at the beginning of an array, at index zero, and shifts any existing elements right by one.

array.insert() can insert a new element at any position in the array. Its `index` parameter is the index where the new element will be added. The element existing at the index used in the function call and any others to its right are shifted one place to the right:



```
//@version=5
indicator("`array.insert()`")
a = array.new_float(5, 0)
for i = 0 to 4
    array.set(a, i, i + 1)
if barstate.islast
    label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a), size = size.large)
    array.insert(a, 2, 999)
    label.new(bar_index, 0, "AFTER\na: " + str.tostring(a), style = label.style_label_u
```

array.push() will add a new element at the end of an array.

## Removing

Four functions can be used to remove elements from an array. The first three will return the value of the removed element.

array.remove() removes the element at the `index` value used, and returns that element's value.

array.shift() removes the first element from an array and returns its value.

array.pop() removes the last element of an array and returns its value.

array.clear() will remove all elements from an array. Note that clearing an array won't delete the underlying data. See the example below which illustrates how this works:

```
//@version=5
indicator("`array.clear()` example", overlay = true)

// We create a label array and add a label to the array on each new bar
var a = array.new_label()
label lbl = label.new(bar_index, high, "Text", color = color.red)
array.push(a, lbl)

var table t = table.new(position.top_right, 1, 1)
// We clear the array on the last bar which won't delete the individual labels
if barstate.islast
    array.clear(a)
    table.cell(t, 0, 0, "Array elements count: " + str.tostring(array.size(a)), bgcolor
```

## Using an array as a stack

Stacks are LIFO (last in, first out) constructions. They behave somewhat like a vertical pile of books to which books can only be added or removed one at a time, always from the top. Pine Script™ arrays can be used as a stack, in which case you will use the array.push() and array.pop() functions to add and remove elements at the end of the array.

`array.push(prices, close)` will add a new element to the end of the `prices` array, increasing the array's size by one.

`array.pop(prices)` will remove the end element from the `prices` array, return its value and decrease the array's size by one.

See how the functions are used here to remember successive lows in rallies:


```
//@version=5
indicator("Lows from new highs", "", true)
var lows = array.new_float(0)
flushLows = false

// Remove last element from the stack when `_cond` is true.
array_pop(id, cond) => cond and array.size(id) > 0 ? array.pop(id) : float(na)

if ta.rising(high, 1)
    // Rising highs; push a new low on the stack.
    array.push(lows, low)
    // Force the return type of this `if` block to be the same as that of the next bloc
    bool(na)
else if array.size(lows) >= 4 or low < array.min(lows)
    // We have at least 4 lows or price has breached the lowest low;
    // sort lows and set flag indicating we will plot and flush the levels.
    array.sort(lows, order.ascending)
    flushLows := true

// If needed, plot and flush lows.
lowLevel = array_pop(lows, flushLows)
plot(lowLevel, "Low 1", low > lowLevel ? color.silver : color.purple, 2, plot.style_li
lowLevel := array_pop(lows, flushLows)
plot(lowLevel, "Low 2", low > lowLevel ? color.silver : color.purple, 3, plot.style_li
lowLevel := array_pop(lows, flushLows)
plot(lowLevel, "Low 3", low > lowLevel ? color.silver : color.purple, 4, plot.style_li
lowLevel := array_pop(lows, flushLows)
plot(lowLevel, "Low 4", low > lowLevel ? color.silver : color.purple, 5, plot.style_li

if flushLows
    // Clear remaining levels after the last 4 have been plotted.
    array.clear(lows)
```

## Using an array as a queue

Queues are FIFO (first in, first out) constructions. They behave somewhat like cars arriving at a red light. New cars are queued at the end of the line, and the first car to leave will be the first one that arrived to the red light.

In the following code example, we let users decide through the script's inputs how many labels they want to have on their chart. We use that quantity to determine the size of the array of labels we then create, initializing the array's elements to `na`.

When a new pivot is detected, we create a label for it, saving the label's ID in the `pLabel` variable. We then queue

the ID of that label by using array.push() to append the new label's ID to the end of the array, making our array size one greater than the maximum number of labels to keep on the chart.

Lastly, we de-queue the oldest label by removing the array's first element using array.shift() and deleting the label referenced by that array element's value. As we have now de-queued an element from our queue, the array contains `pivotCountInput` elements once again. Note that on the dataset's first bars we will be deleting `na` label IDs until the maximum number of labels has been created, but this does not cause runtime errors. Let's look at our code:


```
//@version=5
MAX_LABELS = 100
indicator("Show Last n High Pivots", "", true, max_labels_count = MAX_LABELS)

pivotCountInput = input.int(5, "How many pivots to show", minval = 0, maxval = MAX_LABI
pivotLegsInput  = input.int(3, "Pivot legs", minval = 1, maxval = 5)

// Create an array containing the user-selected max count of label IDs.
var labelIds = array.new_label(pivotCountInput)

pHi = ta.pivothigh(pivotLegsInput, pivotLegsInput)
if not na(pHi)
    // New pivot found; plot its label `i_pivotLegs` bars back.
    pLabel = label.new(bar_index[pivotLegsInput], pHi, str.tostring(pHi, format.mintick
    // Queue the new label's ID by appending it to the end of the array.
    array.push(labelIds, pLabel)
    // De-queue the oldest label ID from the queue and delete the corresponding label.
    label.delete(array.shift(labelIds))
```

## Calculations on arrays

While series variables can be viewed as a horizontal set of values stretching back in time, Pine Script™'s one-dimensional arrays can be viewed as vertical structures residing on each bar. As an array's set of elements is not a time series, Pine Script™'s usual mathematical functions are not allowed on them. Special-purpose functions must be used to operate on all of an array's values. The available functions are: array.abs(), array.avg(), array.covariance(), array.min(), array.max(), array.median(), array.mode(), array.percentile_linear_interpolation(), array.percentile_nearest_rank(), array.percentrank(), array.range(), array.standardize(), array.stdev(), array.sum(), array.variance().

Note that contrary to the usual mathematical functions in Pine Script™, those used on arrays do not return `na` when some of the values they calculate on have `na` values. There are a few exceptions to this rule:

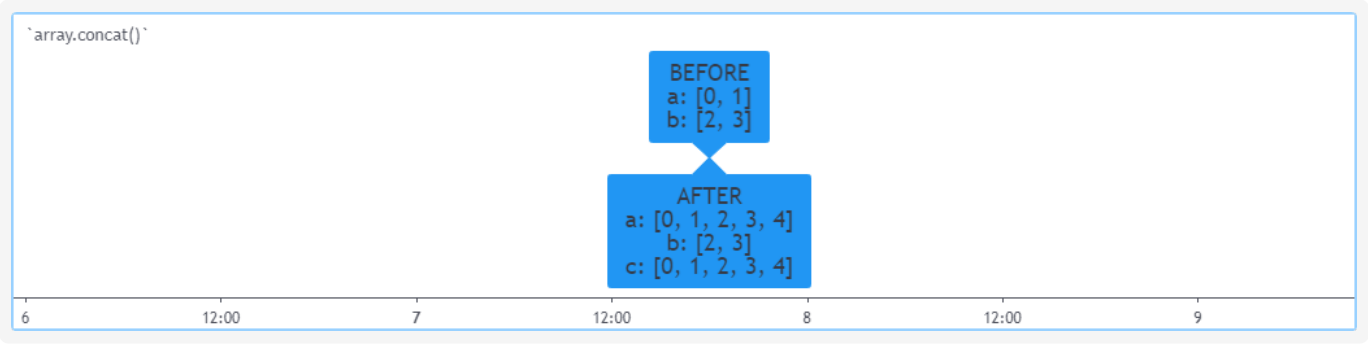
- When all array elements have `na` value or the array contains no elements, `na` is returned.

`array.standardize()` however, will return an empty array.
- `array.mode()` will return `na` when no mode is found.

# Manipulating arrays

## Concatenation

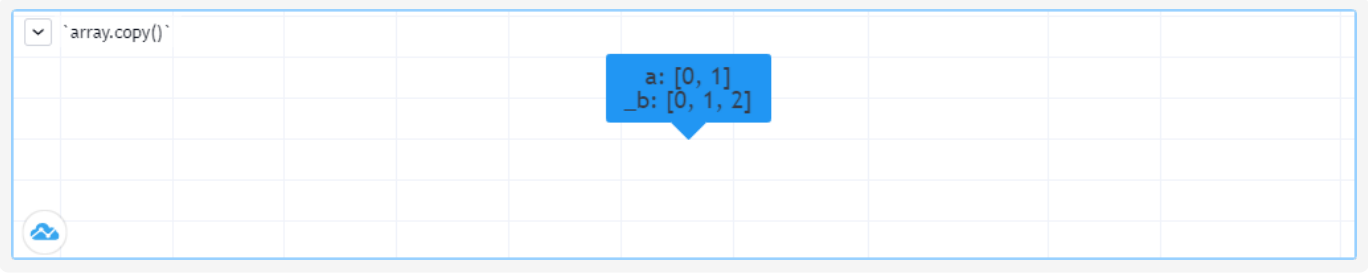
Two arrays can be merged—or concatenated—using array.concat(). When arrays are concatenated, the second array is appended to the end of the first, so the first array is modified while the second one remains intact. The function returns the array ID of the first array:




```
//@version=5
indicator("`array.concat()`")
a = array.new_float(0)
b = array.new_float(0)
array.push(a, 0)
array.push(a, 1)
array.push(b, 2)
array.push(b, 3)
if barstate.islast
    label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a) + "\nb: " + str.tostring(b)
    c = array.concat(a, b)
    array.push(c, 4)
    label.new(bar_index, 0, "AFTER\na: " + str.tostring(a) + "\nb: " + str.tostring(b)
```

## Copying

You can copy an array using array.copy(). Here we copy the array `a` to a new array named `_b`:



```
//@version=5
indicator("`array.copy()`")
a = array.new_float(0)
array.push(a, 0)
array.push(a, 1)
if barstate.islast
    b = array.copy(a)
    array.push(b, 2)
    label.new(bar_index, 0, "a: " + str.tostring(a) + "\nb: " + str.tostring(b), size =
```

Note that simply using `_b = a` in the previous example would not have copied the array, but only its ID. From thereon, both variables would point to the same array, so using either one would affect the same array.

## Joining

Use array.join() to concatenate all of the elements in the array into a string and separate these elements with the specified separator:

```
//@version=5
indicator("")
v1 = array.new_string(10, "test")
v2 = array.new_string(10, "test")
array.push(v2, "test1")
v3 = array.new_float(5, 5)
v4 = array.new_int(5, 5)
l1 = label.new(bar_index, close, array.join(v1))
l2 = label.new(bar_index, close, array.join(v2, ","))
l3 = label.new(bar_index, close, array.join(v3, ","))
l4 = label.new(bar_index, close, array.join(v4, ","))
```

## Sorting

Arrays containing “int” or “float” elements can be sorted in either ascending or descending order using array.sort(). The `order` parameter is optional and defaults to order.ascending. As all `array.*()` function arguments, it is of form “series”, so can be determined at runtime, as is done here. Note that in the example, which array is sorted is also determined at runtime:


```
//@version=5
indicator("`array.sort()`")
a = array.new_float(0)
b = array.new_float(0)
array.push(a, 2)
array.push(a, 0)
array.push(a, 1)
array.push(b, 4)
array.push(b, 3)
array.push(b, 5)
if barstate.islast
    barUp = close > open
    array.sort(barUp ? a : b, barUp ? order.ascending : order.descending)
    label.new(bar_index, 0,
      "a " + (barUp ? "is sorted ▲: "    : "is not sorted: ") + str.tostring(a) + "\n\n"
      "b " + (barUp ? "is not sorted: " : "is sorted ▼: ")    + str.tostring(b), size =
```

Another useful option for sorting arrays is to use the array.sort_indices() function, which takes a reference to the original array and returns an array containing the indices from the original array. Please note that this function won't modify the original array. The `order` parameter is optional and defaults to order.ascending.

## Reversing

Use array.reverse() to reverse an array:

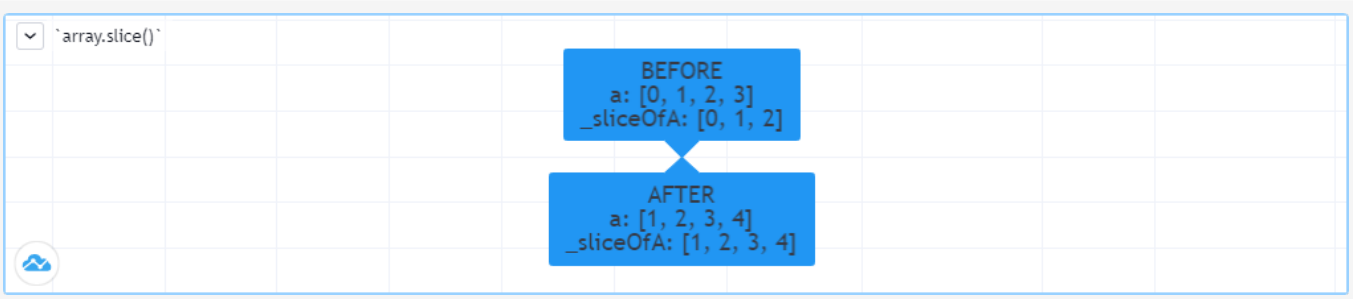
```
//@version=5
indicator("`array.reverse()`")
a = array.new_float(0)
array.push(a, 0)
array.push(a, 1)
array.push(a, 2)
if barstate.islast
    array.reverse(a)
    label.new(bar_index, 0, "a: " + str.tostring(a))
```

# Slicing

Slicing an array using array.slice() creates a shallow copy of a subset of the parent array. You determine the size of the subset to slice using the `index_from` and `index_to` parameters. The `index_to` argument must be one greater than the end of the subset you want to slice.

The shallow copy created by the slice acts like a window on the parent array's content. The indices used for the slice define the window's position and size over the parent array. If, as in the example below, a slice is created from the first three elements of an array (indices 0 to 2), then regardless of changes made to the parent array, and as long as it contains at least three elements, the shallow copy will always contain the parent array's first three elements.

Additionally, once the shallow copy is created, operations on the copy are mirrored on the parent array. Adding an element to the end of the shallow copy, as is done in the following example, will widen the window by one element and also insert that element in the parent array at index 3. In this example, to slice the subset from index 0 to index 2 of array `a`, we must use `_sliceOfA = array.slice(a, 0, 3)`:




```
//@version=5
indicator("`array.slice()`")
a = array.new_float(0)
array.push(a, 0)
array.push(a, 1)
array.push(a, 2)
array.push(a, 3)
if barstate.islast
    // Create a shadow of elements at index 1 and 2 from array `a`.
    sliceOfA = array.slice(a, 0, 3)
    label.new(bar_index, 0, "BEFORE\na: " + str.tostring(a) + "\nsliceOfA: " + str.tost
    // Remove first element of parent array `a`.
    array.remove(a, 0)
    // Add a new element at the end of the shallow copy, thus also affecting the origin
    array.push(sliceOfA, 4)
    label.new(bar_index, 0, "AFTER\na: " + str.tostring(a) + "\nsliceOfA: " + str.tostr
```

# Searching arrays

We can test if a value is part of an array with the array.includes() function, which returns true if the element is found. We can find the first occurrence of a value in an array by using the array.indexof() function. The first occurence is the one with the lowest index. We can also find the last occurrence of a value with array.lastindexof():

```
//@version=5
indicator("Searching in arrays")
valueInput = input.int(1)
a = array.new_float(0)
array.push(a, 0)
array.push(a, 1)
array.push(a, 2)
array.push(a, 1)
if barstate.islast
    valueFound      = array.includes(a, valueInput)
    firstIndexFound = array.indexof(a, valueInput)
    lastIndexFound  = array.lastindexof(a, valueInput)
    label.new(bar_index, 0, "a: " + str.tostring(a) +
      "\nFirst " + str.tostring(valueInput) + (firstIndexFound != -1 ? " value was four
      "\nLast " + str.tostring(valueInput)  + (lastIndexFound  != -1 ? " value was four
```

We can also perform a binary search on an array but note that performing a binary search on an array means that the array will first need to be sorted in ascending order only. The array.binary_search() function will return the value's index if it was found or -1 if it wasn't. If we want to always return an existing index from the array even if our chosen value wasn't found, then we can use one of the other binary search functions available. The array.binary_search_leftmost() function, which returns an index if the value was found or the first index to the left where the value would be found. The array.binary_search_rightmost() function is almost identical and returns an index if the value was found or the first index to the right where the value would be found.

## Error handling

Malformed `array.*()` call syntax in Pine scripts will cause the usual **compiler** error messages to appear in Pine Script™ Editor's console, at the bottom of the window, when you save a script. Refer to the Pine Script™ v5 Reference Manual when in doubt regarding the exact syntax of function calls.

Scripts using arrays can also throw **runtime** errors, which appear in place of the indicator's name on charts. We discuss those runtime errors in this section.

## Index xx is out of bounds. Array size is yy

This will most probably be the most frequent error you encounter. It will happen when you reference an nonexistent array index. The "xx" value will be the value of the faulty index you tried to use, and "yy" will be the size of the array. Recall that array indices start at zero—not one—and end at the array's size, minus one. An array of size 3's last valid index is thus `2`.

To avoid this error, you must make provisions in your code logic to prevent using an index lying outside of the array's index boundaries. This code will generate the error because the last index we use in the loop is outside the valid index range for the array:

```
//@version=5
indicator("Out of bounds index")
a = array.new_float(3)
for i = 1 to 3
    array.set(a, i, i)
plot(array.pop(a))
```

The correct `for` statement is:

```
for i = 0 to 2
```

To loop on all array elements in an array of unknown size, use:

```
//@version=5
indicator("Protected `for` loop")
sizeInput = input.int(0, "Array size", minval = 0, maxval = 100000)
a = array.new_float(sizeInput)
for i = 0 to (array.size(a) == 0 ? na : array.size(a) - 1)
    array.set(a, i, i)
plot(array.pop(a))
```

When you size arrays dynamically using a field in your script's *Settings/Inputs* tab, protect the boundaries of that value using input.int()'s `minval` and `maxval` parameters:

```
//@version=5
indicator("Protected array size")
sizeInput = input.int(10, "Array size", minval = 1, maxval = 100000)
a = array.new_float(sizeInput)
for i = 0 to sizeInput - 1
    array.set(a, i, i)
plot(array.size(a))
```

See the Looping section of this page for more information.

## Cannot call array methods when ID of array is 'na'

When an array ID is initialized to `na`, operations on it are not allowed, since no array exists. All that exists at that point is an array variable containing the `na` value rather that a valid array ID pointing to an existing array. Note that an array created with no elements in it, as you do when you use `a = array.new_int(0)`, has a valid ID nonetheless. This code will throw the error we are discussing:

```
//@version=5
indicator("Out of bounds index")
int[] a = na
array.push(a, 111)
label.new(bar_index, 0, "a: " + str.tostring(a))
```

To avoid it, create an array with size zero using:

```
int[] a = array.new_int(0)
```

or:

```
a = array.new_int(0)
```

## Array is too large. Maximum size is 100000

This error will appear if your code attempts to declare an array with a size greater than 100,000. It will also occur if, while dynamically appending elements to an array, a new element would increase the array's size past the maximum.

## Cannot create an array with a negative size

We haven't found any use for arrays of negative size yet, but if you ever do, we may allow them :)

## Cannot use shift() if array is empty.

This error will occur if array.shift() is called to remove the first element of an empty array.

## Cannot use pop() if array is empty.

This error will occur if array.pop() is called to remove the last element of an empty array.

## Index ‘from’ should be less than index ‘to’

When two indices are used in functions such as array.slice(), the first index must always be smaller than the second one.

## Slice is out of bounds of the parent array

This message occurs whenever the parent array’s size is modified in such a way that it makes the shallow copy created by a slice point outside the boundaries of the parent array. This code will reproduce it because after creating a slice from index 3 to 4 (the last two elements of our five-element parent array), we remove the parent’s first element, making its size four and its last index 3. From that moment on, the shallow copy which is still poiting to the “window” at the parent array’s indices 3 to 4, is pointing out of the parent array’s boundaries:

```
//@version=5
indicator("Slice out of bounds")
a = array.new_float(5, 0)
b = array.slice(a, 3, 5)
array.remove(a, 0)
c = array.indexof(b, 2)
plot(c)
```


User-defined functions

Objects