Multi-dimensional Arrays

Julia, like most technical computing languages, provides a first-class array implementation. Most technical computing languages pay a lot of attention to their array implementation at the expense of other containers. Julia does not treat arrays in any special way. The array library is implemented almost completely in Julia itself, and derives its performance from the compiler, just like any other code written in Julia. As such, it's also possible to define custom array types by inheriting from `AbstractArray`. See the manual section on the AbstractArray interface for more details on implementing a custom array type.

An array is a collection of objects stored in a multi-dimensional grid. In the most general case, an array may contain objects of type `Any`. For most computational purposes, arrays should contain objects of a more specific type, such as `Float64` or `Int32`.

In general, unlike many other technical computing languages, Julia does not expect programs to be written in a vectorized style for performance. Julia's compiler uses type inference and generates optimized code for scalar array indexing, allowing programs to be written in a style that is convenient and readable, without sacrificing performance, and using less memory at times.

In Julia, all arguments to functions are passed by sharing (i.e. by pointers). Some technical computing languages pass arrays by value, and while this prevents accidental modification by callees of a value in the caller, it makes avoiding unwanted copying of arrays difficult. By convention, a function name ending with a `!` indicates that it will mutate or destroy the value of one or more of its arguments (compare, for example, `sort` and `sort!`). Callees must make explicit copies to ensure that they don't modify inputs that they don't intend to change. Many non- mutating functions are implemented by calling a function of the same name with an added `!` at the end on an explicit copy of the input, and returning that copy.

Basic Functions

Function	Description
`eltype(A)`	the type of the elements contained in `A`
`length(A)`	the number of elements in `A`
`ndims(A)`	the number of dimensions of `A`

`size(A)`	a tuple containing the dimensions of `A`
`size(A,n)`	the size of `A` along dimension `n`
`axes(A)`	a tuple containing the valid indices of `A`
`axes(A,n)`	a range expressing the valid indices along dimension `n`
`eachindex(A)`	an efficient iterator for visiting each position in `A`
`stride(A,k)`	the stride (linear index distance between adjacent elements) along dimension `k`
`strides(A)`	a tuple of the strides in each dimension

Construction and Initialization

Many functions for constructing and initializing arrays are provided. In the following list of such functions, calls with a `dims...` argument can either take a single tuple of dimension sizes or a series of dimension sizes passed as a variable number of arguments. Most of these functions also accept a first input `T`, which is the element type of the array. If the type `T` is omitted it will default to `Float64`.

Function	Description
`Array{T}(undef, dims...)`	an uninitialized dense `Array`
`zeros(T, dims...)`	an `Array` of all zeros
`ones(T, dims...)`	an `Array` of all ones
`trues(dims...)`	a `BitArray` with all values `true`
`falses(dims...)`	a `BitArray` with all values `false`
`reshape(A, dims...)`	an array containing the same data as `A`, but with different dimensions
`copy(A)`	copy `A`
`deepcopy(A)`	copy `A`, recursively copying its elements
`similar(A, T, dims...)`	an uninitialized array of the same type as `A` (dense, sparse, etc.), but with the specified element type and dimensions. The second and third arguments are both optional, defaulting to the element type and

	dimensions of `A` if omitted.
`reinterpret(T, A)`	an array with the same binary data as `A`, but with element type `T`
`rand(T, dims...)`	an `Array` with random, iid [1] and uniformly distributed values in the half-open interval $[0, 1)$
`randn(T, dims...)`	an `Array` with random, iid and standard normally distributed values
`Matrix{T}(I, m, n)`	`m`-by-`n` identity matrix. Requires `using LinearAlgebra` for `I`.
`range(start, stop=stop, length=n)`	range of `n` linearly spaced elements from `start` to `stop`
`fill!(A, x)`	fill the array `A` with the value `x`
`fill(x, dims...)`	an `Array` filled with the value `x`

To see the various ways we can pass dimensions to these functions, consider the following examples:

```
julia> zeros(Int8, 2, 3)
2×3 Array{Int8,2}:
 0  0  0
 0  0  0

julia> zeros(Int8, (2, 3))
2×3 Array{Int8,2}:
 0  0  0
 0  0  0

julia> zeros((2, 3))
2×3 Array{Float64,2}:
 0.0  0.0  0.0
 0.0  0.0  0.0
```

Here, `(2, 3)` is a `Tuple` and the first argument — the element type — is optional, defaulting to `Float64`.

Array literals

Arrays can also be directly constructed with square braces; the syntax `[A, B, C, ...]` creates a one dimensional array (i.e., a vector) containing the comma-separated arguments as its elements. The element type (`eltype`) of the resulting array is automatically determined by the types of the arguments

inside the braces. If all the arguments are the same type, then that is its `eltype`. If they all have a common promotion type then they get converted to that type using `convert` and that type is the array's `eltype`. Otherwise, a heterogeneous array that can hold anything — a `Vector{Any}` — is constructed; this includes the literal `[]` where no arguments are given.

```
julia> [1,2,3] # An array of `Int`s
3-element Array{Int64,1}:
 1
 2
 3

julia> promote(1, 2.3, 4//5) # This combination of Int, Float64 and Rational promote
(1.0, 2.3, 0.8)

julia> [1, 2.3, 4//5] # Thus that's the element type of this Array
3-element Array{Float64,1}:
 1.0
 2.3
 0.8

julia> []
Any[]
```

Concatenation

If the arguments inside the square brackets are separated by semicolons (;) or newlines instead of commas, then their contents are *vertically concatenated* together instead of the arguments being used as elements themselves.

```
julia> [1:2, 4:5] # Has a comma, so no concatenation occurs. The ranges are themselv
2-element Array{UnitRange{Int64},1}:
 1:2
 4:5

julia> [1:2; 4:5]
4-element Array{Int64,1}:
 1
 2
 4
 5

julia> [1:2
```

```
       4:5
       6]
5-element Array{Int64,1}:
 1
 2
 4
 5
 6
```

Similarly, if the arguments are separated by tabs or spaces, then their contents are *horizontally concatenated* together.

```
julia> [1:2  4:5  7:8]
2×3 Array{Int64,2}:
 1  4  7
 2  5  8

julia> [[1,2]  [4,5]  [7,8]]
2×3 Array{Int64,2}:
 1  4  7
 2  5  8

julia> [1 2 3] # Numbers can also be horizontally concatenated
1×3 Array{Int64,2}:
 1  2  3
```

Using semicolons (or newlines) and spaces (or tabs) can be combined to concatenate both horizontally and vertically at the same time.

```
julia> [1 2
        3 4]
2×2 Array{Int64,2}:
 1  2
 3  4

julia> [zeros(Int, 2, 2) [1; 2]
        [3 4]            5]
3×3 Array{Int64,2}:
 0  0  1
 0  0  2
 3  4  5
```

More generally, concatenation can be accomplished through the `cat` function. These syntaxes are

shorthands for function calls that themselves are convenience functions:

Syntax	Function	Description
	`cat`	concatenate input arrays along dimension(s) `k`
`[A; B; C; ...]`	`vcat`	shorthand for `cat(A...; dims=1)
`[A B C ...]`	`hcat`	shorthand for `cat(A...; dims=2)
`[A B; C D; ...]`	`hvcat`	simultaneous vertical and horizontal concatenation

Typed array literals

An array with a specific element type can be constructed using the syntax `T[A, B, C, ...]`. This will construct a 1-d array with element type `T`, initialized to contain elements `A`, `B`, `C`, etc. For example, `Any[x, y, z]` constructs a heterogeneous array that can contain any values.

Concatenation syntax can similarly be prefixed with a type to specify the element type of the result.

```
julia> [[1 2] [3 4]]
1×4 Array{Int64,2}:
 1  2  3  4

julia> Int8[[1 2] [3 4]]
1×4 Array{Int8,2}:
 1  2  3  4
```

Comprehensions

Comprehensions provide a general and powerful way to construct arrays. Comprehension syntax is similar to set construction notation in mathematics:

```
A = [ F(x,y,...) for x=rx, y=ry, ... ]
```

The meaning of this form is that `F(x,y,...)` is evaluated with the variables `x`, `y`, etc. taking on each value in their given list of values. Values can be specified as any iterable object, but will commonly be ranges like `1:n` or `2:(n-1)`, or explicit arrays of values like `[1.2, 3.4, 5.7]`. The result is an N-d dense array with dimensions that are the concatenation of the dimensions of the variable ranges `rx`, `ry`, etc. and each `F(x,y,...)` evaluation returns a scalar.

The following example computes a weighted average of the current element and its left and right neighbor along a 1-d grid. :

```
julia> x = rand(8)
8-element Array{Float64,1}:
 0.843025
 0.869052
 0.365105
 0.699456
 0.977653
 0.994953
 0.41084
 0.809411

julia> [ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
6-element Array{Float64,1}:
 0.736559
 0.57468
 0.685417
 0.912429
 0.8446
 0.656511
```

The resulting array type depends on the types of the computed elements just like array literals do. In order to control the type explicitly, a type can be prepended to the comprehension. For example, we could have requested the result in single precision by writing:

```
Float32[ 0.25*x[i-1] + 0.5*x[i] + 0.25*x[i+1] for i=2:length(x)-1 ]
```

Generator Expressions

Comprehensions can also be written without the enclosing square brackets, producing an object known as a generator. This object can be iterated to produce values on demand, instead of allocating an array and storing them in advance (see Iteration). For example, the following expression sums a series without allocating memory:

```
julia> sum(1/n^2 for n=1:1000)
1.6439345666815615
```

When writing a generator expression with multiple dimensions inside an argument list, parentheses are needed to separate the generator from subsequent arguments:

```
julia> map(tuple, 1/(i+j) for i=1:2, j=1:2, [1:4;])
ERROR: syntax: invalid iteration specification
```

All comma-separated expressions after `for` are interpreted as ranges. Adding parentheses lets us add a third argument to `map`:

```
julia> map(tuple, (1/(i+j) for i=1:2, j=1:2), [1 3; 2 4])
2×2 Array{Tuple{Float64,Int64},2}:
 (0.5, 1)       (0.333333, 3)
 (0.333333, 2)  (0.25, 4)
```

Generators are implemented via inner functions. Just like inner functions used elsewhere in the language, variables from the enclosing scope can be "captured" in the inner function. For example, `sum(p[i] - q[i] for i=1:n)` captures the three variables `p`, `q` and `n` from the enclosing scope. Captured variables can present performance challenges; see performance tips.

Ranges in generators and comprehensions can depend on previous ranges by writing multiple `for` keywords:

```
julia> [(i,j) for i=1:3 for j=1:i]
6-element Array{Tuple{Int64,Int64},1}:
 (1, 1)
 (2, 1)
 (2, 2)
 (3, 1)
 (3, 2)
 (3, 3)
```

In such cases, the result is always 1-d.

Generated values can be filtered using the `if` keyword:

```
julia> [(i,j) for i=1:3 for j=1:i if i+j == 4]
2-element Array{Tuple{Int64,Int64},1}:
 (2, 2)
 (3, 1)
```

Indexing

The general syntax for indexing into an n-dimensional array `A` is:

```
X = A[I_1, I_2, ..., I_n]
```

where each `I_k` may be a scalar integer, an array of integers, or any other supported index. This includes `Colon` (`:`) to select all indices within the entire dimension, ranges of the form `a:c` or `a:b:c` to select contiguous or strided subsections, and arrays of booleans to select elements at their `true` indices.

If all the indices are scalars, then the result `X` is a single element from the array `A`. Otherwise, `X` is an array with the same number of dimensions as the sum of the dimensionalities of all the indices.

If all indices `I_k` are vectors, for example, then the shape of `X` would be `(length(I_1), length(I_2), ..., length(I_n))`, with location `i_1, i_2, ..., i_n` of `X` containing the value `A[I_1[i_1], I_2[i_2], ..., I_n[i_n]]`.

Example:

```
julia> A = reshape(collect(1:16), (2, 2, 2, 2))
2×2×2×2 Array{Int64,4}:
[:, :, 1, 1] =
 1  3
 2  4

[:, :, 2, 1] =
 5  7
 6  8

[:, :, 1, 2] =
  9  11
 10  12

[:, :, 2, 2] =
 13  15
 14  16

julia> A[1, 2, 1, 1] # all scalar indices
3

julia> A[[1, 2], [1], [1, 2], [1]] # all vector indices
2×1×2×1 Array{Int64,4}:
[:, :, 1, 1] =
 1
 2

[:, :, 2, 1] =
 5
```

```
 6

julia> A[[1, 2], [1], [1, 2], 1] # a mix of index types
2×1×2 Array{Int64,3}:
[:, :, 1] =
 1
 2

[:, :, 2] =
 5
 6
```

Note how the size of the resulting array is different in the last two cases.

If `I_1` is changed to a two-dimensional matrix, then `X` becomes an `n+1`-dimensional array of shape `(size(I_1, 1), size(I_1, 2), length(I_2), ..., length(I_n))`. The matrix adds a dimension.

Example:

```
julia> A = reshape(collect(1:16), (2, 2, 2, 2));

julia> A[[1 2; 1 2]]
2×2 Array{Int64,2}:
 1  2
 1  2

julia> A[[1 2; 1 2], 1, 2, 1]
2×2 Array{Int64,2}:
 5  6
 5  6
```

The location `i_1, i_2, i_3, ..., i_{n+1}` contains the value at `A[I_1[i_1, i_2], I_2[i_3], ..., I_n[i_{n+1}]]`. All dimensions indexed with scalars are dropped. For example, if `J` is an array of indices, then the result of `A[2, J, 3]` is an array with size `size(J)`. Its `j`th element is populated by `A[2, J[j], 3]`.

As a special part of this syntax, the `end` keyword may be used to represent the last index of each dimension within the indexing brackets, as determined by the size of the innermost array being indexed. Indexing syntax without the `end` keyword is equivalent to a call to `getindex`:

```
X = getindex(A, I_1, I_2, ..., I_n)
```

Example:

```
julia> x = reshape(1:16, 4, 4)
4×4 reshape(::UnitRange{Int64}, 4, 4) with eltype Int64:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> x[2:3, 2:end-1]
2×2 Array{Int64,2}:
 6  10
 7  11

julia> x[1, [2 3; 4 1]]
2×2 Array{Int64,2}:
  5  9
 13  1
```

Indexed Assignment

The general syntax for assigning values in an n-dimensional array `A` is:

```
A[I_1, I_2, ..., I_n] = X
```

where each `I_k` may be a scalar integer, an array of integers, or any other supported index. This includes `Colon` (`:`) to select all indices within the entire dimension, ranges of the form `a:c` or `a:b:c` to select contiguous or strided subsections, and arrays of booleans to select elements at their `true` indices.

If all indices `I_k` are integers, then the value in location `I_1, I_2, ..., I_n` of `A` is overwritten with the value of `X`, `convert`ing to the `eltype` of `A` if necessary.

If any index `I_k` selects more than one location, then the right hand side `X` must be an array with the same shape as the result of indexing `A[I_1, I_2, ..., I_n]` or a vector with the same number of elements. The value in location `I_1[i_1], I_2[i_2], ..., I_n[i_n]` of `A` is overwritten with the value `X[I_1, I_2, ..., I_n]`, converting if necessary. The element-wise assignment operator `.=` may be used to broadcast `X` across the selected locations:

```
A[I_1, I_2, ..., I_n] .= X
```

Just as in Indexing, the `end` keyword may be used to represent the last index of each dimension within the indexing brackets, as determined by the size of the array being assigned into. Indexed assignment syntax without the `end` keyword is equivalent to a call to `setindex!`:

```
setindex!(A, X, I_1, I_2, ..., I_n)
```

Example:

```
julia> x = collect(reshape(1:9, 3, 3))
3×3 Array{Int64,2}:
 1  4  7
 2  5  8
 3  6  9

julia> x[3, 3] = -9;

julia> x[1:2, 1:2] = [-1 -4; -2 -5];

julia> x
3×3 Array{Int64,2}:
 -1  -4   7
 -2  -5   8
  3   6  -9
```

Supported index types

In the expression `A[I_1, I_2, ..., I_n]`, each `I_k` may be a scalar index, an array of scalar indices, or an object that represents an array of scalar indices and can be converted to such by `to_indices`:

1. A scalar index. By default this includes:
   - Non-boolean integers
   - `CartesianIndex{N}`s, which behave like an `N`-tuple of integers spanning multiple dimensions (see below for more details)
2. An array of scalar indices. This includes:
   - Vectors and multidimensional arrays of integers
   - Empty arrays like `[]`, which select no elements
   - Ranges like `a:c` or `a:b:c`, which select contiguous or strided subsections from `a` to `c` (inclusive)
   - Any custom array of scalar indices that is a subtype of `AbstractArray`
   - Arrays of `CartesianIndex{N}` (see below for more details)
3. An object that represents an array of scalar indices and can be converted to such by `to_indices`. By default this includes:

- `Colon()` (`:`), which represents all indices within an entire dimension or across the entire array
- Arrays of booleans, which select elements at their `true` indices (see below for more details)

Some examples:

```
julia> A = reshape(collect(1:2:18), (3, 3))
3×3 Array{Int64,2}:
 1   7  13
 3   9  15
 5  11  17

julia> A[4]
7

julia> A[[2, 5, 8]]
3-element Array{Int64,1}:
  3
  9
 15

julia> A[[1 4; 3 8]]
2×2 Array{Int64,2}:
 1   7
 5  15

julia> A[[]]
Int64[]

julia> A[1:2:5]
3-element Array{Int64,1}:
 1
 5
 9

julia> A[2, :]
3-element Array{Int64,1}:
  3
  9
 15

julia> A[:, 3]
3-element Array{Int64,1}:
 13
 15
 17
```

Cartesian indices

The special `CartesianIndex{N}` object represents a scalar index that behaves like an `N`-tuple of integers spanning multiple dimensions. For example:

```
julia> A = reshape(1:32, 4, 4, 2);

julia> A[3, 2, 1]
7

julia> A[CartesianIndex(3, 2, 1)] == A[3, 2, 1] == 7
true
```

Considered alone, this may seem relatively trivial; `CartesianIndex` simply gathers multiple integers together into one object that represents a single multidimensional index. When combined with other indexing forms and iterators that yield `CartesianIndex`es, however, this can produce very elegant and efficient code. See Iteration below, and for some more advanced examples, see this blog post on multidimensional algorithms and iteration.

Arrays of `CartesianIndex{N}` are also supported. They represent a collection of scalar indices that each span `N` dimensions, enabling a form of indexing that is sometimes referred to as pointwise indexing. For example, it enables accessing the diagonal elements from the first "page" of `A` from above:

```
julia> page = A[:,:,1]
4×4 Array{Int64,2}:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> page[[CartesianIndex(1,1),
             CartesianIndex(2,2),
             CartesianIndex(3,3),
             CartesianIndex(4,4)]]
4-element Array{Int64,1}:
  1
  6
 11
 16
```

This can be expressed much more simply with dot broadcasting and by combining it with a normal integer index (instead of extracting the first `page` from `A` as a separate step). It can even be combined with a `:` to extract both diagonals from the two pages at the same time:

```
julia> A[CartesianIndex.(axes(A, 1), axes(A, 2)), 1]
4-element Array{Int64,1}:
  1
  6
 11
 16

julia> A[CartesianIndex.(axes(A, 1), axes(A, 2)), :]
4×2 Array{Int64,2}:
  1  17
  6  22
 11  27
 16  32
```

> **Warning**
>
> `CartesianIndex` and arrays of `CartesianIndex` are not compatible with the `end` keyword to represent the last index of a dimension. Do not use `end` in indexing expressions that may contain either `CartesianIndex` or arrays thereof.

Logical indexing

Often referred to as logical indexing or indexing with a logical mask, indexing by a boolean array selects elements at the indices where its values are `true`. Indexing by a boolean vector `B` is effectively the same as indexing by the vector of integers that is returned by `findall(B)`. Similarly, indexing by a `N`-dimensional boolean array is effectively the same as indexing by the vector of `CartesianIndex{N}`s where its values are `true`. A logical index must be a vector of the same length as the dimension it indexes into, or it must be the only index provided and match the size and dimensionality of the array it indexes into. It is generally more efficient to use boolean arrays as indices directly instead of first calling `findall`.

```
julia> x = reshape(1:16, 4, 4)
4×4 reshape(::UnitRange{Int64}, 4, 4) with eltype Int64:
 1  5   9  13
 2  6  10  14
 3  7  11  15
 4  8  12  16

julia> x[[false, true, true, false], :]
2×4 Array{Int64,2}:
```

```
 2  6  10  14
 3  7  11  15

julia> mask = map(ispow2, x)
4×4 Array{Bool,2}:
 1  0  0  0
 1  0  0  0
 0  0  0  0
 1  1  0  1

julia> x[mask]
5-element Array{Int64,1}:
  1
  2
  4
  8
 16
```

Number of indices

Cartesian indexing

The ordinary way to index into an `N`-dimensional array is to use exactly `N` indices; each index selects the position(s) in its particular dimension. For example, in the three-dimensional array `A = rand(4, 3, 2)`, `A[2, 3, 1]` will select the number in the second row of the third column in the first "page" of the array. This is often referred to as *cartesian indexing*.

Linear indexing

When exactly one index `i` is provided, that index no longer represents a location in a particular dimension of the array. Instead, it selects the `i`th element using the column-major iteration order that linearly spans the entire array. This is known as *linear indexing*. It essentially treats the array as though it had been reshaped into a one-dimensional vector with `vec`.

```
julia> A = [2 6; 4 7; 3 1]
3×2 Array{Int64,2}:
 2  6
 4  7
 3  1

julia> A[5]
7
```

```
julia> vec(A)[5]
7
```

A linear index into the array `A` can be converted to a `CartesianIndex` for cartesian indexing with `CartesianIndices(A)[i]` (see `CartesianIndices`), and a set of `N` cartesian indices can be converted to a linear index with `LinearIndices(A)[i_1, i_2, ..., i_N]` (see `LinearIndices`).

```
julia> CartesianIndices(A)[5]
CartesianIndex(2, 2)

julia> LinearIndices(A)[2, 2]
5
```

It's important to note that there's a very large assymmetry in the performance of these conversions. Converting a linear index to a set of cartesian indices requires dividing and taking the remainder, whereas going the other way is just multiplies and adds. In modern processors, integer division can be 10-50 times slower than multiplication. While some arrays — like `Array` itself — are implemented using a linear chunk of memory and directly use a linear index in their implementations, other arrays — like `Diagonal` — need the full set of cartesian indices to do their lookup (see `IndexStyle` to introspect which is which). As such, when iterating over an entire array, it's much better to iterate over `eachindex(A)` instead of `1:length(A)`. Not only will the former be much faster in cases where `A` is `IndexCartesian`, but it will also support OffsetArrays, too.

Omitted and extra indices

In addition to linear indexing, an `N`-dimensional array may be indexed with fewer or more than `N` indices in certain situations.

Indices may be omitted if the trailing dimensions that are not indexed into are all length one. In other words, trailing indices can be omitted only if there is only one possible value that those omitted indices could be for an in-bounds indexing expression. For example, a four-dimensional array with size `(3, 4, 2, 1)` may be indexed with only three indices as the dimension that gets skipped (the fourth dimension) has length one. Note that linear indexing takes precedence over this rule.

```
julia> A = reshape(1:24, 3, 4, 2, 1)
3×4×2×1 reshape(::UnitRange{Int64}, 3, 4, 2, 1) with eltype Int64:
[:, :, 1, 1] =
 1  4  7  10
 2  5  8  11
 3  6  9  12

[:, :, 2, 1] =
```

```
 13  16  19  22
 14  17  20  23
 15  18  21  24

julia> A[1, 3, 2] # Omits the fourth dimension (length 1)
19

julia> A[1, 3] # Attempts to omit dimensions 3 & 4 (lengths 2 and 1)
ERROR: BoundsError: attempt to access 3×4×2×1 reshape(::UnitRange{Int64}, 3, 4, 2, 1

julia> A[19] # Linear indexing
19
```

When omitting *all* indices with `A[]`, this semantic provides a simple idiom to retrieve the only element in an array and simultaneously ensure that there was only one element.

Similarly, more than `N` indices may be provided if all the indices beyond the dimensionality of the array are `1` (or more generally are the first and only element of `axes(A, d)` where `d` is that particular dimension number). This allows vectors to be indexed like one-column matrices, for example:

```
julia> A = [8,6,7]
3-element Array{Int64,1}:
 8
 6
 7

julia> A[2,1]
6
```

Iteration

The recommended ways to iterate over a whole array are

```
for a in A
    # Do something with the element a
end

for i in eachindex(A)
    # Do something with i and/or A[i]
end
```

The first construct is used when you need the value, but not index, of each element. In the second

construct, `i` will be an `Int` if `A` is an array type with fast linear indexing; otherwise, it will be a `CartesianIndex`:

```
julia> A = rand(4,3);

julia> B = view(A, 1:3, 2:3);

julia> for i in eachindex(B)
           @show i
       end
i = CartesianIndex(1, 1)
i = CartesianIndex(2, 1)
i = CartesianIndex(3, 1)
i = CartesianIndex(1, 2)
i = CartesianIndex(2, 2)
i = CartesianIndex(3, 2)
```

In contrast with `for i = 1:length(A)`, iterating with `eachindex` provides an efficient way to iterate over any array type.

Array traits

If you write a custom `AbstractArray` type, you can specify that it has fast linear indexing using

```
Base.IndexStyle(::Type{<:MyArray}) = IndexLinear()
```

This setting will cause `eachindex` iteration over a `MyArray` to use integers. If you don't specify this trait, the default value `IndexCartesian()` is used.

Array and Vectorized Operators and Functions

The following operators are supported for arrays:

1. Unary arithmetic – `-`, `+`
2. Binary arithmetic – `-`, `+`, `*`, `/`, `\`, `^`
3. Comparison – `==`, `!=`, ≈ (`isapprox`), ≉

To enable convenient vectorization of mathematical and other operations, Julia provides the dot syntax `f.(args...)`, e.g. `sin.(x)` or `min.(x,y)`, for elementwise operations over arrays or mixtures of arrays and scalars (a Broadcasting operation); these have the additional advantage of "fusing" into a single loop

when combined with other dot calls, e.g. `sin.(cos.(x))`.

Also, *every* binary operator supports a dot version that can be applied to arrays (and combinations of arrays and scalars) in such fused broadcasting operations, e.g. `z .== sin.(x .* y)`.

Note that comparisons such as `==` operate on whole arrays, giving a single boolean answer. Use dot operators like `.==` for elementwise comparisons. (For comparison operations like `<`, *only* the elementwise `.<` version is applicable to arrays.)

Also notice the difference between `max.(a,b)`, which `broadcast`s `max` elementwise over `a` and `b`, and `maximum(a)`, which finds the largest value within `a`. The same relationship holds for `min.(a,b)` and `minimum(a)`.

Broadcasting

It is sometimes useful to perform element-by-element binary operations on arrays of different sizes, such as adding a vector to each column of a matrix. An inefficient way to do this would be to replicate the vector to the size of the matrix:

```
julia> a = rand(2,1); A = rand(2,3);

julia> repeat(a,1,3)+A
2×3 Array{Float64,2}:
 1.20813  1.82068  1.25387
 1.56851  1.86401  1.67846
```

This is wasteful when dimensions get large, so Julia provides `broadcast`, which expands singleton dimensions in array arguments to match the corresponding dimension in the other array without using extra memory, and applies the given function elementwise:

```
julia> broadcast(+, a, A)
2×3 Array{Float64,2}:
 1.20813  1.82068  1.25387
 1.56851  1.86401  1.67846

julia> b = rand(1,2)
1×2 Array{Float64,2}:
 0.867535  0.00457906

julia> broadcast(+, a, b)
2×2 Array{Float64,2}:
 1.71056  0.847604
```

```
 1.73659  0.873631
```

Dotted operators such as `.+` and `.*` are equivalent to `broadcast` calls (except that they fuse, as described above). There is also a `broadcast!` function to specify an explicit destination (which can also be accessed in a fusing fashion by `.=` assignment). In fact, `f.(args...)` is equivalent to `broadcast(f, args...)`, providing a convenient syntax to broadcast any function (dot syntax). Nested "dot calls" `f.(...)` (including calls to `.+` etcetera) automatically fuse into a single `broadcast` call.

Additionally, `broadcast` is not limited to arrays (see the function documentation); it also handles scalars, tuples and other collections. By default, only some argument types are considered scalars, including (but not limited to) `Number`s, `String`s, `Symbol`s, `Type`s, `Function`s and some common singletons like `missing` and `nothing`. All other arguments are iterated over or indexed into elementwise.

```
julia> convert.(Float32, [1, 2])
2-element Array{Float32,1}:
 1.0
 2.0

julia> ceil.(UInt8, [1.2 3.4; 5.6 6.7])
2×2 Array{UInt8,2}:
 0x02  0x04
 0x06  0x07

julia> string.(1:3, ". ", ["First", "Second", "Third"])
3-element Array{String,1}:
 "1. First"
 "2. Second"
 "3. Third"
```

Sometimes, you want a container (like an array) that would normally participate in broadcast to be "protected" from broadcast's behavior of iterating over all of its elements. By placing it inside another container (like a single element `Tuple`) broadcast will treat it as a single value.

```
julia> ([1, 2, 3], [4, 5, 6]) .+ ([1, 2, 3],)
([2, 4, 6], [5, 7, 9])

julia> ([1, 2, 3], [4, 5, 6]) .+ tuple([1, 2, 3])
([2, 4, 6], [5, 7, 9])
```

Implementation

The base array type in Julia is the abstract type `AbstractArray{T,N}`. It is parameterized by the number of dimensions `N` and the element type `T`. `AbstractVector` and `AbstractMatrix` are aliases for the 1-d and 2-d cases. Operations on `AbstractArray` objects are defined using higher level operators and functions, in a way that is independent of the underlying storage. These operations generally work correctly as a fallback for any specific array implementation.

The `AbstractArray` type includes anything vaguely array-like, and implementations of it might be quite different from conventional arrays. For example, elements might be computed on request rather than stored. However, any concrete `AbstractArray{T,N}` type should generally implement at least `size(A)` (returning an `Int` tuple), `getindex(A,i)` and `getindex(A,i1,...,iN)`; mutable arrays should also implement `setindex!`. It is recommended that these operations have nearly constant time complexity, or technically Õ(1) complexity, as otherwise some array functions may be unexpectedly slow. Concrete types should also typically provide a `similar(A,T=eltype(A),dims=size(A))` method, which is used to allocate a similar array for `copy` and other out-of-place operations. No matter how an `AbstractArray{T,N}` is represented internally, `T` is the type of object returned by *integer* indexing (`A[1, ..., 1]`, when `A` is not empty) and `N` should be the length of the tuple returned by `size`. For more details on defining custom `AbstractArray` implementations, see the array interface guide in the interfaces chapter.

`DenseArray` is an abstract subtype of `AbstractArray` intended to include all arrays where elements are stored contiguously in column-major order (see additional notes in Performance Tips). The `Array` type is a specific instance of `DenseArray`; `Vector` and `Matrix` are aliases for the 1-d and 2-d cases. Very few operations are implemented specifically for `Array` beyond those that are required for all `AbstractArray`s; much of the array library is implemented in a generic manner that allows all custom arrays to behave similarly.

`SubArray` is a specialization of `AbstractArray` that performs indexing by sharing memory with the original array rather than by copying it. A `SubArray` is created with the `view` function, which is called the same way as `getindex` (with an array and a series of index arguments). The result of `view` looks the same as the result of `getindex`, except the data is left in place. `view` stores the input index vectors in a `SubArray` object, which can later be used to index the original array indirectly. By putting the `@views` macro in front of an expression or block of code, any `array[...]` slice in that expression will be converted to create a `SubArray` view instead.

`BitArray`s are space-efficient "packed" boolean arrays, which store one bit per boolean value. They can be used similarly to `Array{Bool}` arrays (which store one byte per boolean value), and can be converted to/from the latter via `Array(bitarray)` and `BitArray(array)`, respectively.

An array is "strided" if it is stored in memory with well-defined spacings (strides) between its elements. A strided array with a supported element type may be passed to an external (non-Julia) library like BLAS or LAPACK by simply passing its `pointer` and the stride for each dimension. The `stride(A, d)` is the

distance between elements along dimension `d`. For example, the builtin `Array` returned by `rand(5,7,2)` has its elements arranged contiguously in column major order. This means that the stride of the first dimension — the spacing between elements in the same column — is `1`:

```
julia> A = rand(5,7,2);

julia> stride(A,1)
1
```

The stride of the second dimension is the spacing between elements in the same row, skipping as many elements as there are in a single column (`5`). Similarly, jumping between the two "pages" (in the third dimension) requires skipping `5*7 == 35` elements. The `strides` of this array is the tuple of these three numbers together:

```
julia> strides(A)
(1, 5, 35)
```

In this particular case, the number of elements skipped *in memory* matches the number of *linear indices* skipped. This is only the case for contiguous arrays like `Array` (and other `DenseArray` subtypes) and is not true in general. Views with range indices are a good example of *non-contiguous* strided arrays; consider `V = @view A[1:3:4, 2:2:6, 2:-1:1]`. This view `V` refers to the same memory as `A` but is skipping and re-arranging some of its elements. The stride of the first dimension of `V` is `3` because we're only selecting every third row from our original array:

```
julia> V = @view A[1:3:4, 2:2:6, 2:-1:1];

julia> stride(V, 1)
3
```

This view is similarly selecting every other column from our original `A` — and thus it needs to skip the equivalent of two five-element columns when moving between indices in the second dimension:

```
julia> stride(V, 2)
10
```

The third dimension is interesting because its order is reversed! Thus to get from the first "page" to the second one it must go *backwards* in memory, and so its stride in this dimension is negative!

```
julia> stride(V, 3)
-35
```

This means that the `pointer` for `V` is actually pointing into the middle of `A`'s memory block, and it refers to elements both backwards and forwards in memory. See the interface guide for strided arrays for more details on defining your own strided arrays. `StridedVector` and `StridedMatrix` are convenient aliases for many of the builtin array types that are considered strided arrays, allowing them to dispatch to select specialized implementations that call highly tuned and optimized BLAS and LAPACK functions using just the pointer and strides.

It is worth emphasizing that strides are about offsets in memory rather than indexing. If you are looking to convert between linear (single-index) indexing and cartesian (multi-index) indexing, see `LinearIndices` and `CartesianIndices`.

- 1 *iid*, independently and identically distributed.

« Metaprogramming Missing Values »