Arrays with custom indices

Conventionally, Julia's arrays are indexed starting at 1, whereas some other languages start numbering at 0, and yet others (e.g., Fortran) allow you to specify arbitrary starting indices. While there is much merit in picking a standard (i.e., 1 for Julia), there are some algorithms which simplify considerably if you can index outside the range `1:size(A,d)` (and not just `0:size(A,d)-1`, either). To facilitate such computations, Julia supports arrays with arbitrary indices.

The purpose of this page is to address the question, "what do I have to do to support such arrays in my own code?" First, let's address the simplest case: if you know that your code will never need to handle arrays with unconventional indexing, hopefully the answer is "nothing." Old code, on conventional arrays, should function essentially without alteration as long as it was using the exported interfaces of Julia. If you find it more convenient to just force your users to supply traditional arrays where indexing starts at one, you can add

```
Base.require_one_based_indexing(arrays...)
```

where `arrays...` is a list of the array objects that you wish to check for anything that violates 1-based indexing.

Generalizing existing code

As an overview, the steps are:

- replace many uses of `size` with `axes`
- replace `1:length(A)` with `eachindex(A)`, or in some cases `LinearIndices(A)`
- replace explicit allocations like `Array{Int}(undef, size(B))` with `similar(Array{Int}, axes(B))`

These are described in more detail below.

Things to watch out for

Because unconventional indexing breaks many people's assumptions that all arrays start indexing with 1, there is always the chance that using such arrays will trigger errors. The most frustrating bugs would be incorrect results or segfaults (total crashes of Julia). For example, consider the following function:

```
function mycopy!(dest::AbstractVector, src::AbstractVector)
    length(dest) == length(src) || throw(DimensionMismatch("vectors must match"))
    # OK, now we're safe to use @inbounds, right? (not anymore!)
    for i = 1:length(src)
        @inbounds dest[i] = src[i]
    end
    dest
end
```

This code implicitly assumes that vectors are indexed from 1; if `dest` starts at a different index than `src`, there is a chance that this code would trigger a segfault. (If you do get segfaults, to help locate the cause try running julia with the option `--check-bounds=yes`.)

Using `axes` for bounds checks and loop iteration

`axes(A)` (reminiscent of `size(A)`) returns a tuple of `AbstractUnitRange` objects, specifying the range of valid indices along each dimension of `A`. When `A` has unconventional indexing, the ranges may not start at 1. If you just want the range for a particular dimension `d`, there is `axes(A, d)`.

Base implements a custom range type, `OneTo`, where `OneTo(n)` means the same thing as `1:n` but in a form that guarantees (via the type system) that the lower index is 1. For any new `AbstractArray` type, this is the default returned by `axes`, and it indicates that this array type uses "conventional" 1-based indexing.

For bounds checking, note that there are dedicated functions `checkbounds` and `checkindex` which can sometimes simplify such tests.

Linear indexing (`LinearIndices`)

Some algorithms are most conveniently (or efficiently) written in terms of a single linear index, `A[i]` even if `A` is multi-dimensional. Regardless of the array's native indices, linear indices always range from `1:length(A)`. However, this raises an ambiguity for one-dimensional arrays (a.k.a., `AbstractVector`): does `v[i]` mean linear indexing , or Cartesian indexing with the array's native indices?

For this reason, your best option may be to iterate over the array with `eachindex(A)`, or, if you require the indices to be sequential integers, to get the index range by calling `LinearIndices(A)`. This will return `axes(A, 1)` if A is an AbstractVector, and the equivalent of `1:length(A)` otherwise.

By this definition, 1-dimensional arrays always use Cartesian indexing with the array's native indices. To help enforce this, it's worth noting that the index conversion functions will throw an error if shape indicates a 1-dimensional array with unconventional indexing (i.e., is a `Tuple{UnitRange}` rather than a tuple of `OneTo`). For arrays with conventional indexing, these functions continue to work the same as always.

Using `axes` and `LinearIndices`, here is one way you could rewrite `mycopy!`:

```
function mycopy!(dest::AbstractVector, src::AbstractVector)
    axes(dest) == axes(src) || throw(DimensionMismatch("vectors must match"))
    for i in LinearIndices(src)
        @inbounds dest[i] = src[i]
    end
    dest
end
```

Allocating storage using generalizations of `similar`

Storage is often allocated with `Array{Int}(undef, dims)` or `similar(A, args...)`. When the result needs to match the indices of some other array, this may not always suffice. The generic replacement for such patterns is to use `similar(storagetype, shape)`. `storagetype` indicates the kind of underlying "conventional" behavior you'd like, e.g., `Array{Int}` or `BitArray` or even `dims->zeros(Float32, dims)` (which would allocate an all-zeros array). `shape` is a tuple of `Integer` or `AbstractUnitRange` values, specifying the indices that you want the result to use. Note that a convenient way of producing an all-zeros array that matches the indices of A is simply `zeros(A)`.

Let's walk through a couple of explicit examples. First, if `A` has conventional indices, then `similar(Array{Int}, axes(A))` would end up calling `Array{Int}(undef, size(A))`, and thus return an array. If `A` is an `AbstractArray` type with unconventional indexing, then `similar(Array{Int}, axes(A))` should return something that "behaves like" an `Array{Int}` but with a shape (including indices) that matches `A`. (The most obvious implementation is to allocate an `Array{Int}(undef, size(A))` and then "wrap" it in a type that shifts the indices.)

Note also that `similar(Array{Int}, (axes(A, 2),))` would allocate an `AbstractVector{Int}` (i.e., 1-dimensional array) that matches the indices of the columns of `A`.

Writing custom array types with non-1 indexing

Most of the methods you'll need to define are standard for any `AbstractArray` type, see Abstract Arrays. This page focuses on the steps needed to define unconventional indexing.

Custom `AbstractUnitRange` types

If you're writing a non-1 indexed array type, you will want to specialize `axes` so it returns a `UnitRange`, or (perhaps better) a custom `AbstractUnitRange`. The advantage of a custom type is that it "signals" the allocation type for functions like `similar`. If we're writing an array type for which indexing will start at 0, we likely want to begin by creating a new `AbstractUnitRange`, `ZeroRange`, where `ZeroRange(n)` is equivalent to `0:n-1`.

In general, you should probably *not* export `ZeroRange` from your package: there may be other packages that implement their own `ZeroRange`, and having multiple distinct `ZeroRange` types is (perhaps counterintuitively) an advantage: `ModuleA.ZeroRange` indicates that `similar` should create a `ModuleA.ZeroArray`, whereas `ModuleB.ZeroRange` indicates a `ModuleB.ZeroArray` type. This design allows peaceful coexistence among many different custom array types.

Note that the Julia package CustomUnitRanges.jl can sometimes be used to avoid the need to write your own `ZeroRange` type.

Specializing `axes`

Once you have your `AbstractUnitRange` type, then specialize `axes`:

```
Base.axes(A::ZeroArray) = map(n->ZeroRange(n), A.size)
```

where here we imagine that `ZeroArray` has a field called `size` (there would be other ways to implement this).

In some cases, the fallback definition for `axes(A, d)`:

```
axes(A::AbstractArray{T,N}, d) where {T,N} = d <= N ? axes(A)[d] : OneTo(1)
```

may not be what you want: you may need to specialize it to return something other than `OneTo(1)` when `d > ndims(A)`. Likewise, in `Base` there is a dedicated function `axes1` which is equivalent to `axes(A, 1)` but which avoids checking (at runtime) whether `ndims(A) > 0`. (This is purely a performance optimization.) It is defined as:

```
axes1(A::AbstractArray{T,0}) where {T} = OneTo(1)
axes1(A::AbstractArray) = axes(A)[1]
```

If the first of these (the zero-dimensional case) is problematic for your custom array type, be sure to specialize it appropriately.

Specializing `similar`

Given your custom `ZeroRange` type, then you should also add the following two specializations for `similar`:

```
function Base.similar(A::AbstractArray, T::Type, shape::Tuple{ZeroRange,Vararg{ZeroR
    # body
end

function Base.similar(f::Union{Function,DataType}, shape::Tuple{ZeroRange,Vararg{Ze
    # body
end
```

Both of these should allocate your custom array type.

Specializing `reshape`

Optionally, define a method

```
Base.reshape(A::AbstractArray, shape::Tuple{ZeroRange,Vararg{ZeroRange}}) = ...
```

and you can `reshape` an array so that the result has custom indices.

For objects that mimic AbstractArray but are not subtypes

`has_offset_axes` depends on having `axes` defined for the objects you call it on. If there is some reason you don't have an `axes` method defined for your object, consider defining a method

```
Base.has_offset_axes(obj::MyNon1IndexedArraylikeObject) = true
```

This will allow code that assumes 1-based indexing to detect a problem and throw a helpful error, rather than returning incorrect results or segfaulting julia.

Catching errors

If your new array type triggers errors in other code, one helpful debugging step can be to comment out `@boundscheck` in your `getindex` and `setindex!` implementation. This will ensure that every element access checks bounds. Or, restart julia with `--check-bounds=yes`.

In some cases it may also be helpful to temporarily disable `size` and `length` for your new array type, since code that makes incorrect assumptions frequently uses these functions.

« Proper maintenance and care of multi-threading locks Module loading »