```
# For `AbstractArray`, we do not differentiate between left and right indices:

memsize(A::Array) = sizeof(A)
# hoping that this works for any `AbstractArray` to which it is applied:
memsize(A::AbstractArray) = memsize(parent(A))

"""
    similarstructure_from_indices(T, indleft, indright, A, conjA = :N)

Returns the structure of an object similar to `A` (e.g. `size` for `AbstractArray`
objects)
which has an `eltype` given by `T` and whose left indices correspond to the indices
`indleft` from `op(A)`, and its right indices correspond to the indices `indright`
from
`op(A)`, where `op` is `conj` if `conjA == :C` or does nothing if `conjA == :N`
(default).
"""
similarstructure_from_indices(T::Type, p1::IndexTuple, p2::IndexTuple,
                              A::AbstractArray, CA::Symbol = :N) =
    _similarstructure_from_indices(T, (p1..., p2...), A)

"""
    similarstructure_from_indices(T, indoA, indoB, indleft, indright, A, B, conjA =
:N, conjB= :N)

Returns the structure of an object similar to `A` (e.g. `size` for `AbstractArray`
objects)
which has an `eltype` given by `T` and whose structure corresponds to a selection
of that
of `opA(A)` and `opB(B)` combined. Out of the collection of indices in `indoA` of
`opA(A)`
and `indoB` of `opB(B)`, we construct an object whose left (right) indices
correspond to
indices `indleft` (`indright`) from that collection. Here, `opA` (`opB`) is `conj`
if
`conjA == :C` (`conjB == :C`) or does nothing if `conjA == :N` (`conjB == :N`),
which is
the default).
"""
similarstructure_from_indices(T::Type, poA::IndexTuple, poB::IndexTuple,
                              p1::IndexTuple, p2::IndexTuple,
                              A::AbstractArray, B::AbstractArray,
                              CA::Symbol = :N, CB::Symbol = :N) =
    _similarstructure_from_indices(T, poA, poB, (p1..., p2...), A, B)

"""
    scalar(C)

Returns the single element of a tensor-like object with zero indices or dimensions.
"""
function scalar end

"""
    add!(α, A, conjA, β, C, indleft, indright)
```

```
Implements `C = β*C+α*permute(op(A))` where `A` is permuted such that the left
(right)
indices of `C` correspond to the indices `indleft` (`indright`) of `A`, and `op` is
`conj`
if `conjA == :C` or the identity map if `conjA == :N` (default). Together,
`(indleft..., indright...)` is a permutation of 1 to the number of indices
(dimensions) of
`A`.
"""
add!(α, A::AbstractArray, CA::Symbol, β, C::AbstractArray, indleft::IndexTuple,
        indright::IndexTuple) = add!(α, A, CA, β, C, (indleft..., indright...))

"""
    trace!(α, A, conjA, β, C, indleft, indright, cind1, cind2)

Implements `C = β*C+α*partialtrace(op(A))` where `A` is permuted and partially
traced,
such that the left (right) indices of `C` correspond to the indices `indleft`
(`indright`)
of `A`, and indices `cindA1` are contracted with indices `cindA2`. Furthermore,
`op` is
`conj` if `conjA == :C` or the identity map if `conjA=:N` (default). Together,
`(indleft..., indright..., cind1, cind2)` is a permutation of 1 to the number of
indices
(dimensions) of `A`.
"""
trace!(α, A::AbstractArray, CA::Symbol, β, C::AbstractArray, indleft::IndexTuple,
        indright::IndexTuple, cind1::IndexTuple, cind2::IndexTuple) =
    trace!(α, A, CA, β, C, (indleft..., indright...), cind1, cind2)

"""
    contract!(α, A, conjA, B, conjB, β, C, oindA, cindA, oindB, cindB, indleft,
indright, syms = nothing)

Implements `C = β*C+α*contract(opA(A),opB(B))` where `A` and `B` are contracted,
such that
the indices `cindA` of `A` are contracted with indices `cindB` of `B`. The open
indices
`oindA` of `A` and `oindB` of `B` are permuted such that `C` has left (right)
indices
corresponding to indices `indleft` (`indright`) out of `(oindA..., oindB...)`. The
operation `opA` (`opB`) acts as `conj` if `conjA` (`conjB`) equal `:C` or as the
identity
map if `conjA` (`conjB`) equal `:N`. Together, `(oindA..., cindA...)` is a
permutation of
1 to the number of indices of `A` and `(oindB..., cindB...)` is a permutation of 1
to the
number of indices of `C`. Furthermore, `length(cindA) == length(cindB)`,
`length(oindA)+length(oindB)` equals the number of indices of `C` and `(indleft...,
indright...)` is a permutation of `1` ot the number of indices of `C`.

The final argument `syms` is optional and can be either `nothing`, or a tuple of
three
symbols, which are used to identify temporary objects in the cache to be used for
```

```
permuting
`A`, `B` and `C` so as to perform the contraction as a matrix multiplication.
"""
contract!(α, A::AbstractArray, CA::Symbol, B::AbstractArray, CB::Symbol,
        β, C::AbstractArray, oindA::IndexTuple, cindA::IndexTuple,
oindB::IndexTuple,
        cindB::IndexTuple, indleft::IndexTuple, indright::IndexTuple, syms =
nothing) =
    contract!(α, A, CA, B, CB, β, C,
                oindA, cindA, oindB, cindB, (indleft..., indright...), syms)

# actual implementations for AbstractArray with ind = (indleft..., indright...)
_similarstructure_from_indices(T, ind, A::AbstractArray) = map(n->size(A, n), ind)

function _similarstructure_from_indices(T, poA::IndexTuple, poB::IndexTuple,
        ind::IndexTuple, A::AbstractArray, B::AbstractArray)

    oszA = map(n->size(A,n), poA)
    oszB = map(n->size(B,n), poB)
    sz = let osz = (oszA..., oszB...)
        map(n->osz[n], ind)
    end
    return sz
end

scalar(C::AbstractArray) = ndims(C)==0 ? C[] : throw(DimensionMismatch())

function add!(α, A::AbstractArray{<:Any, N}, CA::Symbol,
        β, C::AbstractArray{<:Any, N}, indCinA) where {N}

    N == length(indCinA) || throw(IndexError("Invalid permutation of length $N:
$indCinA"))
    if CA == :N
        if isbitstype(eltype(A)) && isbitstype(eltype(C))
            @unsafe_strided A C _add!(α, A, β, C, (indCinA...,))
        else
            _add!(α, StridedView(A), β, StridedView(C), (indCinA...,))
        end
    elseif CA == :C
        if isbitstype(eltype(A)) && isbitstype(eltype(C))
            @unsafe_strided A C _add!(α, conj(A), β, C, (indCinA...,))
        else
            _add!(α, conj(StridedView(A)), β, StridedView(C), (indCinA...,))
        end
    elseif CA == :A
        if isbitstype(eltype(A)) && isbitstype(eltype(C))
            @unsafe_strided A C _add!(α, map(adjoint, A), β, C, (indCinA...,))
        else
            _add!(α, map(adjoint, StridedView(A)), β, StridedView(C), (indCinA...,))
        end
    else
        throw(ArgumentError("Unknown conjugation flag: $CA"))
    end
    return C
end
```

```
_add!(α, A::AbstractStridedView{<:Any,N},
        β, C::AbstractStridedView{<:Any,N}, indCinA::IndexTuple{N}) where N =
    LinearAlgebra.axpby!(α, permutedims(A, indCinA), β, C)

function trace!(α, A::AbstractArray{<:Any, NA}, CA::Symbol, β,
C::AbstractArray{<:Any, NC},
        indCinA, cindA1, cindA2) where {NA,NC}

    NC == length(indCinA) ||
        throw(IndexError("Invalid selection of $NC out of $NA: $indCinA"))
    NA-NC == 2*length(cindA1) == 2*length(cindA2) ||
        throw(IndexError("invalid number of trace dimension"))
    if CA == :N
        if isbitstype(eltype(A)) && isbitstype(eltype(C))
            @unsafe_strided A C _trace!(α, A, β, C,
                (indCinA...,), (cindA1...,), (cindA2...,))
        else
            _trace!(α, StridedView(A), β, StridedView(C),
                (indCinA...,), (cindA1...,), (cindA2...,))
        end
    elseif CA == :C
        if isbitstype(eltype(A)) && isbitstype(eltype(C))
            @unsafe_strided A C _trace!(α, conj(A), β, C,
                (indCinA...,), (cindA1...,), (cindA2...,))
        else
            _trace!(α, conj(StridedView(A)), β, StridedView(C),
                (indCinA...,), (cindA1...,), (cindA2...,))
        end
    elseif CA == :A
        if isbitstype(eltype(A)) && isbitstype(eltype(C))
            @unsafe_strided A C _trace!(α, map(adjoint, A), β, C,
                (indCinA...,), (cindA1...,), (cindA2...,))
        else
            _trace!(α, map(adjoint, StridedView(A)), β, StridedView(C),
                (indCinA...,), (cindA1...,), (cindA2...,))
        end
    else
        throw(ArgumentError("Unknown conjugation flag: $CA"))
    end
    return C
end

function _trace!(α, A::AbstractStridedView,
        β, C::AbstractStridedView, indCinA::IndexTuple{NC},
        cindA1::IndexTuple{NT}, cindA2::IndexTuple{NT}) where {NC,NT}

    sizeA = i->size(A, i)
    strideA = i->stride(A, i)
    tracesize = sizeA.(cindA1)
    tracesize == sizeA.(cindA2) || throw(DimensionMismatch("non-matching trace
sizes"))
    size(C) == sizeA.(indCinA) || throw(DimensionMismatch("non-matching sizes"))

    newstrides = (strideA.(indCinA)..., (strideA.(cindA1) .+ strideA.(cindA2))...)
    newsize = (size(C)..., tracesize...)
```

```
    if A isa UnsafeStridedView
        A2 = UnsafeStridedView(A.ptr, newsize, newstrides, A.offset, A.op)
    else
        A2 = StridedView(A.parent, newsize, newstrides, A.offset, A.op)
    end

    if α != 1
        if β == 0
            Strided._mapreducedim!(x->α*x, +, zero, newsize, (C, A2))
        elseif β == 1
            Strided._mapreducedim!(x->α*x, +, nothing, newsize, (C, A2))
        else
            Strided._mapreducedim!(x->α*x, +, y->β*y, newsize, (C, A2))
        end
    else
        if β == 0
            return Strided._mapreducedim!(identity, +, zero, newsize, (C, A2))
        elseif β == 1
            Strided._mapreducedim!(identity, +, nothing, newsize, (C, A2))
        else
            Strided._mapreducedim!(identity, +, y->β*y, newsize, (C, A2))
        end
    end
    return C
end

function contract!(α, A::AbstractArray, CA::Symbol, B::AbstractArray, CB::Symbol,
        β, C::AbstractArray,
        oindA::IndexTuple, cindA::IndexTuple, oindB::IndexTuple, cindB::IndexTuple,
        indCinoAB::IndexTuple, syms::Union{Nothing, NTuple{3,Symbol}} = nothing)

    TC = eltype(C)
    ipC = TupleTools.invperm(indCinoAB)
    oindAinC = TupleTools.getindices(ipC, _trivtuple(oindA))
    oindBinC = TupleTools.getindices(ipC, length(oindA) .+ _trivtuple(oindB))
    if use_blas() && TC <: BlasFloat
        # check if it is beneficial to change the role of A and B
        ibc = isblascontractable
        lA = length(A)
        lB = length(B)
        lC = length(C)
        memcost1 = lA*(!ibc(A, oindA, cindA, CA) || eltype(A) !== TC) +
                    lB*(!ibc(B, cindB, oindB, CB) || eltype(B) !== TC) +
                    lC*(!ibc(C, oindAinC, oindBinC, :D))
        memcost2 = lB*(!ibc(B, oindB, cindB, CB) || eltype(B) !== TC) +
                    lA*(!ibc(A, cindA, oindA, CA) || eltype(A) !== TC) +
                    lC*(!ibc(C, oindBinC, oindAinC, :D))

        if memcost1 > memcost2
            indCinoBA = let N₁ = length(oindA), N₂ = length(oindB)
                map(n->ifelse(n>N₁, n-N₁, n+N₂), indCinoAB)
            end
            return contract!(α, B, CB, A, CA, β, C,
                                oindB, cindB, oindA, cindA, indCinoBA, syms)
        end
```

```
    end

    pA = (oindA...,cindA...)
    (length(pA) == ndims(A) && TupleTools.isperm(pA)) ||
        throw(IndexError("invalid permutation of length $(ndims(A)): $pA"))
    pB = (oindB...,cindB...)
    (length(pB) == ndims(B) && TupleTools.isperm(pB)) ||
        throw(IndexError("invalid permutation of length $(ndims(B)): $pB"))
    (length(oindA) + length(oindB) == ndims(C)) ||
        throw(IndexError("non-matching output indices in contraction"))
    (ndims(C) == length(indCinoAB) && isperm(indCinoAB)) ||
        throw(IndexError("invalid permutation of length $(ndims(C)): $indCinoAB"))

    sizeA = size(A)
    sizeB = size(B)
    sizeC = size(C)

    csizeA = TupleTools.getindices(sizeA, cindA)
    csizeB = TupleTools.getindices(sizeB, cindB)
    osizeA = TupleTools.getindices(sizeA, oindA)
    osizeB = TupleTools.getindices(sizeB, oindB)

    csizeA == csizeB ||
        throw(DimensionMismatch("non-matching sizes in contracted dimensions"))
    TupleTools.getindices((osizeA..., osizeB...), indCinoAB) == size(C) ||
        throw(DimensionMismatch("non-matching sizes in uncontracted dimensions"))

    if use_blas() && TC <: BlasFloat
        if isblascontractable(A, oindA, cindA, CA) && eltype(A) == TC
            A2 = A
            CA2 = CA
        else
            if syms === nothing
                A2 = similar_from_indices(TC, oindA, cindA, A, CA)
            else
                A2 = cached_similar_from_indices(syms[1], TC, oindA, cindA, A, CA)
            end
            add!(1, A, CA, 0, A2, oindA, cindA)
            CA2 = :N
            oindA = _trivtuple(oindA)
            cindA = _trivtuple(cindA) .+ length(oindA)
        end
        if isblascontractable(B, cindB, oindB, CB) && eltype(B) == TC
            B2 = B
            CB2 = CB
        else
            if syms === nothing
                B2 = similar_from_indices(TC, cindB, oindB, B, CB)
            else
                B2 = cached_similar_from_indices(syms[2], TC, cindB, oindB, B, CB)
            end
            add!(1, B, CB, 0, B2, cindB, oindB)
            CB2 = :N
            cindB = _trivtuple(cindB)
            oindB = _trivtuple(oindB) .+ length(cindB)
```

```
        end
        ipC = TupleTools.invperm(indCinoAB)
        oindAinC = TupleTools.getindices(ipC, _trivtuple(oindA))
        oindBinC = TupleTools.getindices(ipC, length(oindA) .+ _trivtuple(oindB))
        if isblascontractable(C, oindAinC, oindBinC, :D)
            C2 = C
            _blas_contract!(α, A2, CA2, B2, CB2, β, C2,
                            oindA, cindA, oindB, cindB, oindAinC, oindBinC,
                            osizeA, csizeA, osizeB, csizeB)
        else
            if syms === nothing
                C2 = similar_from_indices(TC, oindAinC, oindBinC, C, :N)
            else
                C2 = cached_similar_from_indices(syms[3], TC, oindAinC, oindBinC,
C, :N)
            end
            _blas_contract!(1, A2, CA2, B2, CB2, 0, C2,
                            oindA, cindA, oindB, cindB,
                            _trivtuple(oindA), length(oindA) .+
_trivtuple(oindB),
                            osizeA, csizeA, osizeB, csizeB)

            add!(α, C2, :N, β, C, indCinoAB, ())
        end
    else
        _native_contract!(α, A, CA, B, CB, β, C, oindA, cindA, oindB, cindB,
indCinoAB,
                          osizeA, csizeA, osizeB, csizeB)
    end
    return C
end

function isblascontractable(A::AbstractArray, p1::IndexTuple, p2::IndexTuple,
        C::Symbol)

    eltype(A) <: LinearAlgebra.BlasFloat || return false
    @unsafe_strided A isblascontractable(A, p1, p2, C)
end

function isblascontractable(A::AbstractStridedView, p1::IndexTuple, p2::IndexTuple,
        C::Symbol)

    eltype(A) <: LinearAlgebra.BlasFloat || return false
    sizeA = size(A)
    stridesA = strides(A)
    sizeA1 = TupleTools.getindices(sizeA, p1)
    sizeA2 = TupleTools.getindices(sizeA, p2)
    stridesA1 = TupleTools.getindices(stridesA, p1)
    stridesA2 = TupleTools.getindices(stridesA, p2)

    canfuse1, d1, s1 = _canfuse(sizeA1, stridesA1)
    canfuse2, d2, s2 = _canfuse(sizeA2, stridesA2)

    if C == :D # destination
        return A.op == identity && canfuse1 && canfuse2 && s1 == 1
```

```
    elseif (C == :C && A.op == identity) || (C == :N && A.op == conj)# conjugated
        return canfuse1 && canfuse2 && s2 == 1
    else
        return canfuse1 && canfuse2 && (s1 == 1 || s2 == 1)
    end
end

_canfuse(::Dims{0}, ::Dims{0}) = true, 1, 1
_canfuse(dims::Dims{1}, strides::Dims{1}) = true, dims[1], strides[1]
function _canfuse(dims::Dims{N}, strides::Dims{N}) where {N}
    if dims[1] == 0
        return true, 0, 1
    elseif dims[1] == 1
        return _canfuse(Base.tail(dims), Base.tail(strides))
    else
        b, d, s = _canfuse(Base.tail(dims), Base.tail(strides))
        if b && (s == dims[1]*strides[1] || d == 1)
            dnew = dims[1]*d
            return true, dnew, (dnew == 0 || dnew == 1) ? 1 : strides[1]
        else
            return false, dims[1]*d, strides[1]
        end
    end
end
_trivtuple(t::NTuple{N}) where {N} = ntuple(identity, Val(N))

function _blas_contract!(α, A::AbstractArray, CA, B::AbstractArray, CB,
        β, C::AbstractArray, oindA, cindA, oindB, cindB, oindAinC, oindBinC,
        osizeA, csizeA, osizeB, csizeB)

    @unsafe_strided A B C begin
        A2 = sreshape(permutedims(A, (oindA..., cindA...)), (prod(osizeA),
prod(csizeA)))
        B2 = sreshape(permutedims(B, (cindB..., oindB...)), (prod(csizeB),
prod(osizeB)))
        C2 = sreshape(permutedims(C, (oindAinC..., oindBinC...)),
                (prod(osizeA), prod(osizeB)))
        if CA == :N && CB == :N
            mul!(C2, A2, B2, α, β)
        elseif (CA == :C || CA == :A) && CB == :N
            mul!(C2, conj(A2), B2, α, β)
        elseif CA == :N && (CB == :C || CB == :A)
            mul!(C2, A2, conj(B2), α, β)
        elseif (CA == :C || CA == :A) && (CB == :C || CB == :A)
            mul!(C2, conj(A2), conj(B2), α, β)
        else
            throw(ArgumentError("unknown conjugation flag $CA and $CB"))
        end
    end
    return C
end

function _native_contract!(α, A::AbstractArray, CA::Symbol, B::AbstractArray,
CB::Symbol,
        β, C::AbstractArray, oindA, cindA, oindB, cindB, indCinoAB,
```

```
        osizeA, csizeA, osizeB, csizeB)

    ipC = TupleTools.invperm(indCinoAB)
    if CA == :N
        opA = identity
    elseif CA == :C
        opA = conj
    elseif CA == :A
        opA = adjoint
    else
        throw(ArgumentError("unknown conjugation flag $CA"))
    end
    if CB == :N
        opB = identity
    elseif CB == :C
        opB = conj
    elseif CB == :A
        opB = adjoint
    else
        throw(ArgumentError("unknown conjugation flag $CB"))
    end

    let opA = opA, opB = opB, α = α
        AS = sreshape(permutedims(StridedView(A), (oindA..., cindA...)),
            (osizeA..., one.(osizeB)..., csizeA...))
        BS = sreshape(permutedims(StridedView(B), (oindB..., cindB...)),
            (one.(osizeA)..., osizeB..., csizeB...))
        CS = sreshape(permutedims(StridedView(C), ipC),
            (osizeA..., osizeB..., one.(csizeA)...))
        tsize = (osizeA..., osizeB..., csizeA...)
        if α != 1
            op1 = (x,y) -> α*opA(x)*opB(y)
            if β == 0
                Strided._mapreducedim!(op1, +, zero, tsize, (CS, AS, BS))
            elseif β == 1
                Strided._mapreducedim!(op1, +, nothing, tsize, (CS, AS, BS))
            else
                Strided._mapreducedim!(op1, +, y->β*y, tsize, (CS, AS, BS))
            end
        else
            op2 = (x,y) -> opA(x)*opB(y)
            if β == 0
                if isbitstype(eltype(C))
                    Strided._mapreducedim!(op2, +, zero, tsize, (CS, AS, BS))
                else
                    fill!(C, zero(eltype(C)))
                    Strided._mapreducedim!(op2, +, nothing, tsize, (CS, AS, BS))
                end
            elseif β == 1
                Strided._mapreducedim!(op2, +, nothing, tsize, (CS, AS, BS))
            else
                Strided._mapreducedim!(op2, +, y->β*y, tsize, (CS, AS, BS))
            end
        end
    end
```

```
    return C
end
```