```
"""
    struct GenericRepresentationSpace{G<:Sector} <: RepresentationSpace{G}

Generic implementation of a representation space, i.e. a complex Euclidean space
with a
direct sum structure corresponding to different superselection sectors of type
`G<:Sector`,
e.g. the irreps of a compact or finite group, or the labels of a unitary fusion
category.

This fallback is used when `IteratorSize(values(G)) == IsInfinite()`.
"""
struct GenericRepresentationSpace{G<:Sector} <: RepresentationSpace{G}
    dims::SectorDict{G,Int}
    dual::Bool
end

function GenericRepresentationSpace{G}(dims; dual::Bool = false) where {G<:Sector}
    d = SectorDict{G,Int}()
    for (c, dc) in dims
        k = convert(G, c)
        haskey(d, k) && throw(ArgumentError("Sector $k appears multiple times"))
        !iszero(dc) && push!(d, k=>dc)
    end
    return GenericRepresentationSpace{G}(d, dual)
end
GenericRepresentationSpace{G}(; dual::Bool = false) where {G<:Sector} =
    GenericRepresentationSpace{G}(); dual = dual)
GenericRepresentationSpace{G}(d1::Pair; dual::Bool = false) where {G<:Sector} =
    GenericRepresentationSpace{G}((d1, ); dual = dual)
GenericRepresentationSpace{G}(d1::Pair, dims::Vararg{Pair};
                              dual::Bool = false) where {G<:Sector} =
    GenericRepresentationSpace{G}((d1, dims...); dual = dual)

Base.:(==)(V1::GenericRepresentationSpace, V2::GenericRepresentationSpace) =
    keys(V1.dims) == keys(V2.dims) &&
    values(V1.dims) == values(V2.dims) &&
    V1.dual == V2.dual

Base.hash(V::GenericRepresentationSpace, h::UInt) = hash(V.dual, hash(V.dims, h))

"""
    struct FiniteRepresentationSpace{G<:Sector,N} <: AbstractRepresentationSpace{G}

Optimized implementation for a representation space (fusion category) with a
finite number of labels (simple objects), i.e. a complex Euclidean space with a
direct sum structure corresponding to different superselection sectors of type
`G<:Sector`,
e.g. the irreps of a finite group, or the labels of a unitary fusion category.

This fallback is used when `IteratorSize(values(G))` is of type `HasLength` or
`HasShape`.
"""
struct FiniteRepresentationSpace{G<:Sector,N} <: RepresentationSpace{G}
```

```
    dims::NTuple{N,Int}
    dual::Bool
    function FiniteRepresentationSpace{G,N}(dims::Dims{N}, dual::Bool) where
{G<:Sector,N}
        return new{G,N}(dims, dual)
    end
end

function FiniteRepresentationSpace{G}(dims; dual::Bool = false) where {G<:Sector}
    N = length(values(G))
    d = ntuple(n->0, StaticLength(N))
    isset = ntuple(n->false, StaticLength(N))
    for (c,dc) in dims
        i = findindex(values(G), convert(G, c))
        isset[i] && throw(ArgumentError("Sector $c appears multiple times"))
        isset = TupleTools.setindex(isset, true, i)
        d = TupleTools.setindex(d, dc, i)
    end
    return FiniteRepresentationSpace{G,N}(d, dual)
end
FiniteRepresentationSpace{G}(; dual::Bool = false) where {G<:Sector} =
    FiniteRepresentationSpace{G}((); dual = dual)
FiniteRepresentationSpace{G}(d1::Pair; dual::Bool = false) where {G<:Sector} =
    FiniteRepresentationSpace{G}((d1,); dual = dual)
FiniteRepresentationSpace{G}(d1::Pair, dims::Vararg{Pair};
                                dual::Bool = false) where {G<:Sector} =
    FiniteRepresentationSpace{G}((d1, dims...); dual = dual)
# get rid of N parameter
function FiniteRepresentationSpace{G,N}(args...; dual::Bool = false) where
{G<:Sector, N}
    @assert N == length(values(G))
    FiniteRepresentationSpace{G}(args...; dual = dual)
end

# Never write GenericRepresentationSpace, just use RepresentationSpace
function RepresentationSpace{G}(args...; dual::Bool = false) where {G<:Sector}
    if Base.IteratorSize(values(G)) === IsInfinite()
        GenericRepresentationSpace{G}(args...; dual = dual)
    else
        FiniteRepresentationSpace{G}(args...; dual = dual)
    end
end
RepresentationSpace(dims::Tuple{Vararg{Pair{G,Int}}};
                        dual::Bool = false) where {G<:Sector} =
    RepresentationSpace{G}(dims; dual = dual)
RepresentationSpace(dims::Vararg{Pair{G,Int}}; dual::Bool = false) where
{G<:Sector} =
    RepresentationSpace{G}(dims; dual = dual)
RepresentationSpace(dims::AbstractDict{G,Int}; dual::Bool = false) where
{G<:Sector} =
    RepresentationSpace{G}(dims; dual = dual)
# not inferrable
RepresentationSpace(g::Base.Generator; dual::Bool = false) =
    RepresentationSpace(g...; dual = dual)
```

```
RepresentationSpace(g::AbstractDict; dual::Bool = false) =
    RepresentationSpace(g...; dual = dual)

Base.getindex(::ComplexNumbers, G::Type{<:Sector}) = RepresentationSpace{G}
Base.getindex(::ComplexNumbers, d1::Pair{G,Int}, dims::Pair{G,Int}...) where
{G<:Sector} =
    RepresentationSpace{G}(d1, dims...)

# Corresponding methods:
# properties
dim(V::GenericRepresentationSpace) =
    mapreduce(c->dim(c)*V.dims[c], +, keys(V.dims); init =
zero(dim(one(sectortype(V)))))
dim(V::GenericRepresentationSpace{G}, c::G) where {G<:Sector} =
    get(V.dims, isdual(V) ? dual(c) : c, 0)

dim(V::FiniteRepresentationSpace{G}) where {G<:Sector} =
    reduce(+, dc*dim(c) for (dc,c) in zip(V.dims, values(G));
            init = zero(dim(one(sectortype(V)))))
dim(V::FiniteRepresentationSpace{G}, c::G) where {G<:Sector} =
    V.dims[findindex(values(G), isdual(V) ? dual(c) : c)]

sectors(V::GenericRepresentationSpace{G}) where {G<:Sector} =
    SectorSet{G}(s->isdual(V) ? dual(s) : s, keys(V.dims))
sectors(V::FiniteRepresentationSpace{G,N}) where {G<:Sector,N} =
    SectorSet{G}(Iterators.filter(n->V.dims[n]!=0, 1:N)) do n
        isdual(V) ? dual(values(G)[n]) : values(G)[n]
    end

hassector(V::RepresentationSpace{G}, s::G) where {G<:Sector} = dim(V, s) != 0

Base.conj(V::GenericRepresentationSpace{G}) where {G<:Sector} =
    GenericRepresentationSpace{G}(V.dims, !V.dual)
Base.conj(V::FiniteRepresentationSpace{G,N}) where {G<:Sector,N} =
    FiniteRepresentationSpace{G,N}(V.dims, !V.dual)
isdual(V::RepresentationSpace) = V.dual

# equality / comparison
Base.:(==)(V1::RepresentationSpace, V2::RepresentationSpace) =
    (V1.dims == V2.dims) && V1.dual == V2.dual

# axes
Base.axes(V::RepresentationSpace) = Base.OneTo(dim(V))
function Base.axes(V::RepresentationSpace{G}, c::G) where {G}
    offset = 0
    for c′ in sectors(V)
        c′ == c && break
        offset += dim(c′)*dim(V, c′)
    end
    return (offset+1):(offset+dim(c)*dim(V, c))
end

# Specific constructors for Z_N
const ZNSpace{N} = FiniteRepresentationSpace{ZNIrrep{N},N}
```

```
ZNSpace{N}(dims::NTuple{N,Int}; dual::Bool = false) where {N} = ZNSpace{N}(dims,
dual)
ZNSpace{N}(dims::Vararg{Int,N}; dual::Bool = false) where {N} = ZNSpace{N}(dims,
dual)
ZNSpace(dims::NTuple{N,Int}; dual::Bool = false) where {N} = ZNSpace{N}(dims, dual)
ZNSpace(dims::Vararg{Int,N}; dual::Bool = false) where {N} = ZNSpace{N}(dims, dual)

const ParitySpace = ZNSpace{2}
const ℤ₂Space = ZNSpace{2}
const ℤ₃Space = ZNSpace{3}
const ℤ₄Space = ZNSpace{4}
const U₁Space = GenericRepresentationSpace{U₁}
const CU₁Space = GenericRepresentationSpace{CU₁}
const SU₂Space = GenericRepresentationSpace{SU₂}

# non-Unicode alternatives
const Z2Space = ℤ₂Space
const Z3Space = ℤ₃Space
const Z4Space = ℤ₄Space
const U1Space = U₁Space
const CU1Space = CU₁Space
const SU2Space = SU₂Space

Base.oneunit(::Type{<:RepresentationSpace{G}}) where {G<:Sector} =
    RepresentationSpace{G}(one(G)=>1)

function ⊕(V1::RepresentationSpace{G}, V2::RepresentationSpace{G}) where
{G<:Sector}
    dual1 = isdual(V1)
    dual1 == isdual(V2) ||
        throw(SpaceMismatch("Direct sum of a vector space and a dual space does
not exist"))
    dims = SectorDict{G,Int}()
    for c in union(sectors(V1), sectors(V2))
        cout = ifelse(dual1, dual(c), c)
        dims[cout] = dim(V1,c) + dim(V2,c)
    end
    return RepresentationSpace{G}(dims; dual = dual1)
end

function flip(V::RepresentationSpace)
    if isdual(V)
        typeof(V)((c=>dim(V,c) for c in sectors(V))...)
    else
        typeof(V)((dual(c)=>dim(V,c) for c in sectors(V))...)'
    end
end

function fuse(V1::RepresentationSpace{G}, V2::RepresentationSpace{G}) where
{G<:Sector}
    dims = SectorDict{G,Int}()
    for a in sectors(V1), b in sectors(V2)
        for c in a ⊗ b
            dims[c] = get(dims, c, 0) + Nsymbol(a,b,c)*dim(V1,a)*dim(V2,b)
```

```
            end
        end
        return RepresentationSpace{G}(dims)
    end

    function infinum(V1::RepresentationSpace{G}, V2::RepresentationSpace{G}) where {G}
        if V1.dual == V2.dual
            RepresentationSpace{G}(c=>min(dim(V1,c), dim(V2,c)) for c in
                union(sectors(V1), sectors(V2)), dual = V1.dual)
        else
            throw(SpaceMismatch("Infinum of space and dual space does not exist"))
        end
    end

    function supremum(V1::RepresentationSpace{G}, V2::RepresentationSpace{G}) where {G}
        if V1.dual == V2.dual
            RepresentationSpace{G}(c=>max(dim(V1,c), dim(V2,c)) for c in
                union(sectors(V1), sectors(V2)), dual = V1.dual)
        else
            throw(SpaceMismatch("Supremum of space and dual space does not exist"))
        end
    end

    Base.show(io::IO, ::Type{ℤ₂Space}) = print(io, "ℤ₂Space")
    Base.show(io::IO, ::Type{ℤ₃Space}) = print(io, "ℤ₃Space")
    Base.show(io::IO, ::Type{ℤ₄Space}) = print(io, "ℤ₄Space")
    Base.show(io::IO, ::Type{U₁Space}) = print(io, "U₁Space")
    Base.show(io::IO, ::Type{CU₁Space}) = print(io, "CU₁Space")
    Base.show(io::IO, ::Type{SU₂Space}) = print(io, "SU₂Space")

    function Base.show(io::IO, V::RepresentationSpace{G}) where {G<:Sector}
        show(io, typeof(V))
        print(io, "(")
        seperator = ""
        comma = ", "
        io2 = IOContext(io, :typeinfo => G)
        for c in sectors(V)
            if isdual(V)
                print(io2, seperator, dual(c), "=>", dim(V, c))
            else
                print(io2, seperator, c, "=>", dim(V, c))
            end
            seperator = comma
        end
        print(io, ")")
        V.dual && print(io, "'")
        return nothing
    end
```