

Example: Tagged indices

► Motivation

- Many programs deal with multidimensional arrays
- Mistakes involving misplaced indices are frequent

```
for (std::size_t i = 0; i < L; ++i)
```

```
    for (std::size_t j = 0; j < M; ++j)
```

```
        for (std::size_t k = 0; k < N; ++k)
```

```
            c[i][j] += a[i][k] * b[k][j]; // maybe b[j][k] - who knows ???
```

- In addition, a shorter form of for-loop would be nice

```
for (auto i : L)
```

```
    for (auto j : M)
```

```
        for (auto k : N)
```

```
            c[i][j] += a[i][k] * b[k][j]; // maybe b[j][k] ???
```

- ▶ Replacing for loops with range-based loops

```
for (auto i : L)
```

```
    for (auto j : M)
```

```
        for (auto k : N)
```

```
            c[i][j] += a[i][k] * b[k][j]; // maybe b[j][k] ???
```

- ▶ Can we do it the same way as with single-dimensional containers?

- We need multiple layers of range-like interfaces
 - And the ability to descend in arbitrary order (transposition)
- We need to iterate through multiple arrays at the same time

```
for ( auto& [ci, ai] : std::views::zip( c, a) )
```

```
    for ( auto& [cij, bj] : std::views::zip( cij, b.transpose()) )
```

```
        for ( auto& [aik, bkj] : std::views::zip( ai, bj) )
```

```
            cij += aik * bkj;
```

- `std::views::zip` does not check for equal lengths
- Multiple iterators are incremented in each iteration (compared to one index)
- Would a mathematician like this mess?

► Iterating through virtual ranges

```
for (auto i : std::views::iota(0, 1) )  
  for (auto j : std::views::iota(0, m) )  
    for (auto k : std::views::iota(0, n) )  
      c[i][j] += a[i][k] * b[k][j];
```

- `iota(a,b)` produces a virtual range containing values `a, a+1, ..., b-1`
- the interface may be made nice by publishing `iota` objects instead of the sizes

```
auto L = std::views::iota(0, 1);  
auto M = std::views::iota(0, m);  
auto N = std::views::iota(0, n);
```

```
for (auto i : L)  
  for (auto j : M)  
    for (auto k : N)  
      c[i][j] += a[i][k] * b[k][j]; // maybe b[j][k] ???
```

- All the L/M/N ranges are of the same type - no compile-time checks possible

▶ Distinguishing dimensions

- the interface may be made nice by publishing iota objects instead of the constants

```
constexpr auto L = tagged_range<dim_i>(0, 1);
```

```
constexpr auto M = tagged_range<dim_j>(0, m);
```

```
constexpr auto N = tagged_range<dim_k>(0, n);
```

```
for (auto i : L)
```

```
    for (auto j : M)
```

```
        for (auto k : N)
```

```
            c[i][j] += a[i][k] * b[k][j]; // maybe b[j][k] ???
```

- `tagged_range<T>` class is a virtual range containing values of type `tagged_index<T>`
- Now the tensors a/b/c may require properly tagged indexes in their operator[]
 - The tensors must be declared with the same tags

```
tagged_tensor<float, dim_i, dim_k> a;
```

```
tagged_tensor<float, dim_k, dim_j> b;
```

```
tagged_tensor<float, dim_i, dim_j> c;
```

▶ Declaring tags

- Tags are template arguments - either constants or types
 - Keeping types distinct is far easier than for constants
- It is good to be able to verify that a type is a tag
 - Let all tags derive from a common base struct

```
struct dimension_tag_base {};
```

```
template< typename T> concept dimension_tag = std::derived_from< T, dimension_tag_base>;
```

- Tag declarations

```
struct dim_i : dimension_tag_base {};
```

```
struct dim_j : dimension_tag_base {};
```

```
struct dim_k : dimension_tag_base {};
```

- Generic tensor

```
template< typename ELEMENT, dimension_tag ... TAGS>
```

```
class tagged_tensor;
```

▶ Tagged tensor

- If indices are tagged, the tensor knows which dimension they apply to
 - No need to explicit transposition
 - Requires that tags for a given tensor are unique

```
template< typename ELEMENT, dimension_tag ... TAGS>
```

```
class tagged_tensor {
```

```
public:
```

```
    template< dimension_tag T>
```

```
    tagged_tensor_view< ELEMENT, remove_tag_from_list< T, TAGS...>>
```

```
        operator[]( tagged_index< T> i );
```

```
};
```

- The result of operator[] shall be a view - a reference to a subset of tensor elements.

- ▶ Specifying tensor sizes

- ranges

```
auto L = tagged_range<dim_i>(0, 1);
```

```
auto M = tagged_range<dim_j>(0, m);
```

- the tensor

```
tagged_tensor<float, dim_i, dim_j> c( L, M);
```

- a nicer syntax

```
auto c = make_tagged_tensor<float>( L, M);
```

- ▶ In many cases, a dimension has a fixed size

```
auto W = tagged_range<dim_w>(0, w);
```

```
auto H = tagged_range<dim_h>(0, h);
```

```
constexpr auto C = tagged_range<dim_c>(0, 3);
```

```
auto image = make_tagged_tensor<std::uint8_t>(W, H, C);
```

- operator[] involves multiplication by dimension sizes

- if the sizes are known to the compiler, the multiplication may be faster
- the above interface does not offer this possibility

- ▶ In many cases, a dimension has a fixed size

- We want the compiler to be able to extract the range

- However, the range is passed as a run-time argument to functions

```
void multiply( /*...*/, a_range auto K, a_range auto L, a_range auto M);
```

- It would be inconvenient to have 8 variants of multiply with different signatures
 - The constants must be embedded in the type, not in the value
 - Therefore, we need the concept a_range instead of the type tagged_range

```
constexpr auto three = fixed_tagged_range<dim_c, 0, 3>();
```

```
auto matrix33 = make_tagged_tensor<float>( three, three);
```

```
multiply(/*...*/, three, three, three);
```

- the functions can, at compile time

- determine that the argument is a fixed range
 - extract the constants from the type
 - make_tagged_tensor can produce a type parameterized by these constants

```
tagged_tensor<float, fixed< dim_c, 0, 3>, fixed< dim_c, 0, 3>>
```

- some functions on the types, e.g. size(), may be made conditionally constexpr