Tensor Library - Einstein's notation

The previous tensor library was augmented with the ability to perform tensor operations using **Einstein's notation**.

To do that, index types were defined in a namespace `index` (inside the tensor namespace) and some operators were overloaded. Some operations return a proxy object that can be converted to the corret tensor type.

DYNAMIC RANK TENSORS

Inside the namespace, there is a class `index_t` with a public constructor that accepts in input a `size_t` value to initialize the only data member `const size_t value`. Inside the namespace there are also some declarations of constant indexes used in Einstein's notation operations, each one initialized passing a different integer value, like:
- `const index_t i(0);`
- `const index_t j(1);`
- `const index_t k(2);`

Since dynamic rank tensor does not need a proxy object to be returned because all operations and checks are performed at runtime, to perform tensor operations using Einstein's notation, class tensor was edited by adding a vector `dynamic::indexes_notation_type indexesNotation;` where `indexes_notation_type` is a `typedef std::vector<size_t>` declared in the policy of dynamic rank tensor, which stores the indexes passed as argument in `operator ()`.
To the dynamic tensor class were added also several auxiliary functions to perform the operations below.
(*Also in dynamic rank tensor could be possibile to create a proxy tensor like in fixed rank tensor but we would show two different ways to deal with Einstein's notation problem*).

Operators overloaded:

`operator ()`:

Performs the contraction of one or more indexes repeated, if they are specified, inside a single tensor.
Accepts in input a constant vector of indexes of type `index_t`, like this: `const std::vector<index::index_t>& indexes`.
Returns a new dynamic tensor of type `tensor<T, dynamic>` if contraction is performed, `*this` otherwise.
The complete signature of the function is `tensor<T, dynamic> operator() (const std::vector<index::index_t>& indexes);`.
It is possible to cast statically the outgoing tensor to to a dynamic tensor.

Example:
```
tensor::tensor<int> a4({3,3});
```

```
tensor::tensor<int> trace_result = static_cast<tensor::tensor<int>>
(a4({tensor::index::i, tensor::index::i}));
```

If it performs the contraction of all indexes, it will returns a tensor with rank and dimension equal to 1 that can be casted to a type `T` just by using the overloaded cast operator `operator T()`.

`operator *`:

Performs the contraction of one or more indexes repeated, if they are specified, between two tensors.
Accepts in input a dynamic tensor like `tensor<T,dynamic> t`.
Returns a new dynamic tensor like `tensor<T,dynamic>` if contraction is performed; an empty tensor `tensor<T,dynamic> nothing_tensor({0});` otherwise (if there are no common indexes to be summed).
The complete signature of the function is `tensor<T,dynamic> operator * (tensor<T,dynamic> t);`.

Example:

```
tensor::tensor<int> a2(2,3,5);
tensor::tensor<int> b2(2,3,4);
tensor::tensor<int> c2(5,4);

c2 = a2({tensor::index::k, tensor::index::l, tensor::index::n}) * b2({tensor::index::k, tensor::index::l, tensor::index::m});
```

`operator +`:

Performs the summation between `*this` and the tensor passed as argument `tensor<T,dynamic> t` over the same indexes.
Returns a new dynamic tensor like `tensor<T,dynamic>`. It is necessary to specify all the indexes of the tensor to perform this operation and make sure that the dimension of each one corresponds to the dimension of all indexes in the other tensor.
The complete signature of the function is `tensor<T,dynamic> operator + (tensor<T,dynamic> t);`.

It calls an auxiliary function that deals with both `operator +` and `operator -` at the same time, by passing one more parameter (a `char` value) corresponding to the operation to be performed (`tensor<T,dynamic> operators_plus_minus_temp(tensor<T,dynamic> t, const char sign)`).

`operator -`:

Performs the subtraction between `*this` and the tensor passed as argument `tensor<T,dynamic> t` over the same indexes.
Returns a new dynamic tensor like `tensor<T,dynamic>`. It is necessary to specify all the indexes of the tensor to perform this operation and make sure that the dimension of each one corresponds to the dimension of all indexes in the other tensor.
The complete signature of the function is `tensor<T,dynamic> operator - (tensor<T,dynamic> t);`.

It calls an auxiliary function that deals with both `operator +` and `operator -` at the same time, by passing one more parameter (a `char` value) corresponding to the operation to be performed (`tensor<T,dynamic> operators_plus_minus_temp(tensor<T,dynamic> t, const char sign)`).

Example:

```
tensor::tensor<int> a3({2,3});
tensor::tensor<int> b3({3,2});

tensor::tensor<int> c3 = a3({tensor::index::i, tensor::index::j}) +/-
b3({tensor::index::j, tensor::index::i});
```

FIXED RANK TENSORS

Inside the namespace, there is a struct `index_tr` parameterized on `template <int n>` and inside it a `static constexpr int value = n;` initialized with the template argument.
Inside the namespace there are also some declarations of indexes used in Einstein's notation operations, each one initialized passing a different integer value, like:
- `index_tr<0> i_r;`
- `index_tr<1> j_r;`
- `index_tr<2> k_r;`

Since fixed rank tensor maintains a lot of static information and some operations are performed at compile-time (`operator *`), it is necessary to create a **proxy object** that will be returned by the operation performed.
So, to perform tensor operations using Einstein's notation, proxy tensor class was created and declared as `class proxy_tensor;` parameterized on `template <typename T, size_t R, int ... indexes>` and inherits from fixed rank tensor.
So, the complete definition of the proxy tensor class is:
`class proxy_tensor<T, rank<R>, index::index_tr<indexes>...> : tensor<T, rank<R>>`.

Inside it, there is a constructor (that calls the constructor of fixed rank tensor base class) and a `typename rank<sizeof...(indexes)>::index_type indexesNotation;` that is a vector which stores the indexes passed as argument in the template. To the proxy tensor class were added also several functions (and auxiliary functions) to perform the operations below.

The rank of tensor to be returned in `operator ()` and `operator *` is computed at compile-time.
To do this **templates** were used:
- a `struct my_set;` parameterized with `template <int... Ns>` that represents a set of integer values and could contains a sequence of them in its template argument.
- a `struct set_find;` parameterized with `template <int T, class S>` that recursively stores in its `static constexpr size_t value` variable the values `0` if element `T` is not in set `S` or `1` otherwise.
- a `struct num_same_indexes;` parameterized with `template <class A, int... T>` that recursively stores in its `static constexpr size_t value` the number of indexes that are repeated in a single tensor (used for `operator ()`).
- a `struct num_matched_indexes;` parameterized with `template <class F, class S>` that recursively stores in its `static constexpr size_t value` the number of indexes that are repeated between two different tensors (used for `operator *`).
- a `struct num_same_grouped_indexes;` parameterized with `template <class A, int... T>` that recursively stores in its `static constexpr size_t value` the number of groups of in which indexes are equal, so the repeated indexes in a tensor (used for `operator()`).

So, to determine the rank of the tensor to be returned in the different operators it is necessary to perform in advance a call to these `struct` like this for the `operator ()`:

```
template <int... idxInt>
tensor<T,rank<R - num_same_indexes<my_set<>, idxInt...>::value>> operator()
(index::index_tr<idxInt>... inds);
```

Or like this for the `operator *`:

```
template<size_t R2, int ... second_indexes>
tensor<T, rank<R + R2 - num_matched_indexes<my_set<indexes...>,
my_set<second_indexes...>>::value - num_matched_indexes<my_set<second_indexes...>,
my_set<indexes...>>::value>> operator * (proxy_tensor<T, rank<R2>,
index::index_tr<second_indexes>...> s);
```

Operators overloaded:

`operator ()`:

Performs the contraction of one or more indexes repeated, if they are specified, inside a single tensor.
It is parameterized on `template <int... idxInt>` and accepts in input a sequence (variadic) of `index_tr` which are parameterized on the template argument pack `idxInt`. They correspond to the indexes that will be expanded with the `...` operator: `index::index_tr<idxInt>... argsInt`.
Returns a new tensor with the exact rank (computed at compile-time) desired by the contraction operation, if it is performed. Otherwise, it returns a copy of the original tensor.
The complete signature of the function is `tensor<T, rank<R - num_same_indexes<my_set<>, idxInt...>::value>> operator() (index::index_tr<idxInt>... inds);`.

Example:

```
std::vector<size_t> dimensions = {2,3,4,2};
tensor::tensor<int, tensor::rank<4>> fixed_rank(dimensions);

tensor::index::index_tr<1> a_fixed;
tensor::index::index_tr<2> b_fixed;
tensor::index::index_tr<1> c_fixed;
tensor::index::index_tr<8> d_fixed;

tensor::tensor<int, tensor::rank<2>> t_prova_fixed = fixed_rank(a_fixed, b_fixed,
c_fixed, d_fixed);
```

`operator *`:

Performs the contraction of one or more indexes repeated, if they are specified, between two tensors.
This operator is inside the proxy tensor class because it is necessary to know at compile-time the resulting rank of the operation and to do that we need to specify in the template arguments both the set of indexes of the tow tensors.
So, to perform this operator between two fixed rank tensors it is necessary to create two proxy tensors and then apply the operator `*`.

Accepts in input a proxy tensor like `proxy_tensor<T, rank<R2>, index::index_tr<second_indexes>...> s`.
Returns a new fixed tensor with the exact rank computed at compile-time if contraction is performed, otherwise an assert fails and notice it to the user.

Example:
```
tensor::proxy_tensor<int, tensor::rank<3>, tensor::index::index_tr<1>, tensor::index::index_tr<2>, tensor::index::index_tr<3>> first_proxy_tensor({2,3,4});
tensor::proxy_tensor<int, tensor::rank<2>, tensor::index::index_tr<4>, tensor::index::index_tr<2>> second_proxy_tensor({5,3});

tensor::tensor<int, tensor::rank<3>> tensor_product_result = first_proxy_tensor * second_proxy_tensor;
```

`operator +`:

Performs the summation between `*this` and the tensor passed as argument `tensor<T,rank<R>> t` over the same indexes.
Returns a new fixed rank tensor: `tensor<T,rank<R>>`. It is necessary to specify all the indexes of the tensor to perform this operation and make sure that the dimension of each one corresponds to the dimension of all indexes in the other tensor.
The complete signature of the function is `tensor<T,rank<R>> operator + (const tensor<T, rank<R>>& t);`.

It calls an auxiliary function that deals with both `operator +` and `operator -` at the same time, by passing one more parameter (a `char` value) corresponding to the operation to be performed (`tensor<T, rank<R>> operators_plus_minus_temp(tensor<T,rank<R>> t, const char sign)`).

`operator -`:

Performs the subtraction between `*this` and the tensor passed as argument `tensor<T,rank<R>> t` over the same indexes.
Returns a new fixed rank tensor: `tensor<T,rank<R>>`. It is necessary to specify all the indexes of the tensor to perform this operation and make sure that the dimension of each one corresponds to the dimension of all indexes in the other tensor.
The complete signature of the function is `tensor<T,rank<R>> operator - (const tensor<T, rank<R>>& t);`.

It calls an auxiliary function that deals with both `operator +` and `operator -` at the same time, by passing one more parameter (a `char` value) corresponding to the operation to be performed (`tensor<T, rank<R>> operators_plus_minus_temp(tensor<T,rank<R>> t, const char sign)`).

Example:
```
std::vector<size_t> dimensions1 = {2,3,4};
tensor::tensor<int, tensor::rank<3>> frt1(dimensions1);
```

```
std::vector<size_t> dimensions2 = {4,3,2};
tensor::tensor<int, tensor::rank<3>> frt2(dimensions2);

tensor::index::index_tr<1> ind_fix1;
tensor::index::index_tr<2> ind_fix2;
tensor::index::index_tr<3> ind_fix3;

tensor::tensor<int, tensor::rank<3>> frt_result = frt1(ind_fix1,ind_fix2,ind_fix3) + frt2(ind_fix3,ind_fix2,ind_fix1);
```

RANK 1 TENSORS

To perform tensor operations using Einstein's notation, class tensor was edited by adding a variable `size_t indexNotation;` which stores the index of type `index_t` passed as argument in `operator ()`.

Operators overloaded:

`operator ()`:

Stores the index passed as argument in `indexNotation` variable.
Accepts in input a constant index of type `const index::index_t index` (or an array with only one element).
Returns a new rank 1 tensor of type `tensor<T,rank<1>>` (`*this`).
The complete signature of the function is `tensor<T,rank<1>> operator() (const index::index_t index);`.

`operator *`:

Performs the contractions between `*this` and the tensor passed as argument `tensor<T,rank<1>> t` over the same index.
It checks that the Einstein index of the two rank 1 tensors is the same and it must be `index_t i = 0` (the first index available).
Returns a new rank 1 tensor which dimension is 1 like `tensor<T,rank<1>> result(1);`.
The complete signature of the function is `tensor<T,rank<1>> operator * (tensor<T,rank<1>> t);`.

`operator +`:

Performs the summation between `*this` and the tensor passed as argument `tensor<T,rank<1>> t` over the same index.
Returns a new rank 1 tensor which dimension is the only dimension of the tensor: `tensor<T,rank<1>> result(width.at(0));`.
The complete signature of the function is `tensor<T,rank<1>> operator + (tensor<T,rank<1>> t);`.

It calls an auxiliary function that deals with both `operator +` and `operator -` at the same time, by passing one more parameter corresponding to the operation to be performed (`tensor<T,rank<1>> operators_plus_minus_temp(tensor<T,rank<1>> t, const char sign)`).

`operator -`:

Performs the subtraction between `*this` and the tensor passed as argument `tensor<T,rank<1>> t` over the same index.
Returns a new rank 1 tensor which dimension is the only dimension of the tensor: `tensor<T,rank<1>> result(width.at(0));`.
The complete signature of the function is `tensor<T,rank<1>> operator - (tensor<T,rank<1>> t);`.

It calls an auxiliary function that deals with both `operator +` and `operator -` at the same time, by passing one more parameter corresponding to the operation to be performed (`tensor<T,rank<1>> operators_plus_minus_temp(tensor<T,rank<1>> t, const char sign)`).

Complete example:

```
tensor::tensor<int,tensor::rank<1>> aa(2);
tensor::tensor<int,tensor::rank<1>> bb(2);

tensor::tensor<int,tensor::rank<1>> rr_x =
static_cast<tensor::tensor<int,tensor::rank<1>>> (aa(tensor::index::i) *
bb(tensor::index::i));
tensor::tensor<int,tensor::rank<1>> rr_plus =
static_cast<tensor::tensor<int,tensor::rank<1>>> (aa(tensor::index::i) +
bb(tensor::index::i));
tensor::tensor<int,tensor::rank<1>> rr_minus =
static_cast<tensor::tensor<int,tensor::rank<1>>> (aa(tensor::index::i) -
bb(tensor::index::i));
```

COMPILING

Compile with `g++ tensor.cc -o tensor -std=c++14`.
Then execute `./tensor`.