Luca Settimo - 872778
Luca Fortin - 858986
Master Degree in Computer Science - Data Management and Analytics
[CM0470] Advanced Algorithms and Programming Methods 2
Tensor Project

Tensor Project

Summary

Introduction to the project

This project consists in building a templated **library** handling **tensors**.
The components for this project are **Luca Settimo** (872778@stud.unive.it) and **Luca Fortin** (858986@stud.unive.it).

The programming language requested is **C++** and the code editor used is the Windows version of **Visual Studio Code**.
In order to compile and execute the code we have chosen two different ways:

- Open the **Ubuntu 20.04** terminal into Visual Studio code and compile the project by inserting the the following string:
  - ***g++ -std=c++17 main.cpp -o main***

  Then the executable will be runned:
  - ***./main***
- Installing the **C/C++ extension** for Visual Studio Code and exploiting the **MinGW** installation. This method lets us to run the program in **Windows** directly from the code editor

This program has been developed following the documentation and rules of the **c++17** version, therefore the library produced may not be compatible for versions lower than this one.
The reason for this choice is because, from this version, the use of the keyword **constexpr** after an “if” lets us apply a *static* “if” (an “if” that works at compile time).
“If constexpr” evaluates constexpr expression at compile time and then discards the code in one of the branches:

- This lets us to check whether if a number has a *floating point type* or an *integral type* with only a short instruction

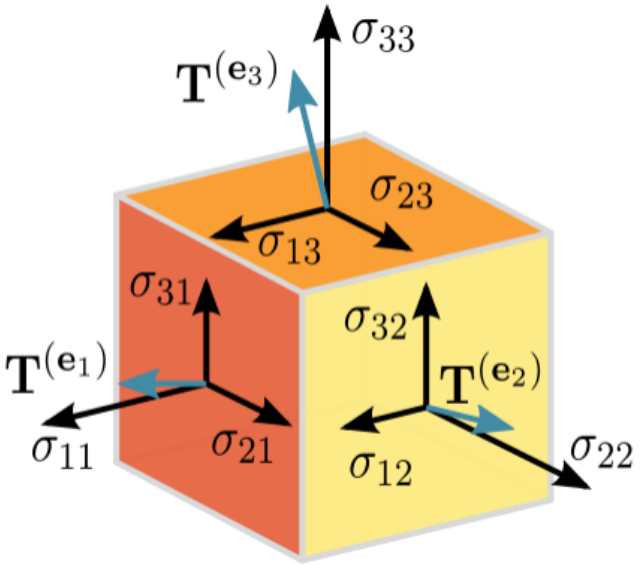
Tensors

Theoretical definition

A **tensor** is a **multidimensional array** which composes a unique data structure used for containing the data values. [1] [2]

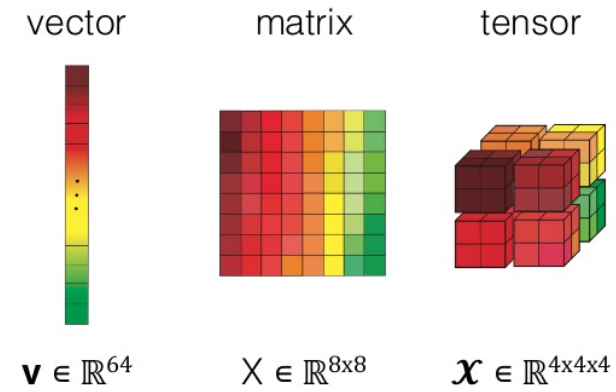
The tensor data structure is composed by:

- **Rank**: corresponds to the number of dimensions of this data structure, for this reason it corresponds to the number of indexes needed for extracting elements.
  The most simple rank numbers are: [2]
  - ***Rank = 0***: it corresponds to a single element in which the dimensionality of the data structure is equal to zero
  - ***Rank = 1***: it corresponds to a one-dimensional vector. The most famous data structure which has a one-dimensional space as an assumption is the array.
  - ***Rank = 2***: it can be represented by a matrix (two-dimensional array) composed by rows and columns
  - ***Rank = 3 and Above***: Usually we refer as “tensor” only when we have a rank equal or greater than 3. Each step up in ranking means that you’re dealing with an extra layer of information in the container (if we see tensors as containers). For example, in geometrical terms, a 3-D tensor is a cube of elements:



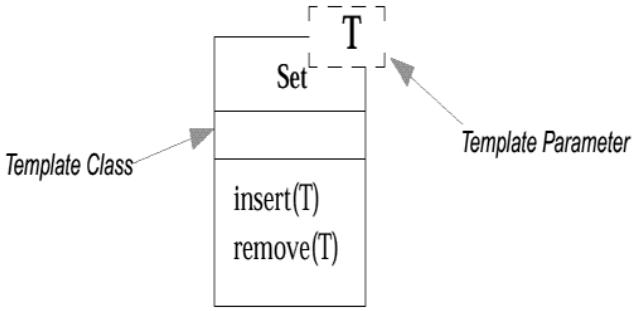
Example of a 3-D tensor [2]

For this library it was requested to develop a handler library that can handle any number of ranks that is equal to 0 or higher.
We also need to manage tensors that have compile-time rank and type, and also just the type and not the rank.



Example of distribution of 64 elements into tensors of rank 1, 2 and 3 [3]

- **Type**: fundamental component of the tensor always required at compile time. The value of this type corresponds to the type of each element. In this project the delivery requires the presence of a class having a **parametric type**, i.e. the presence of a generic type which allows the generalization of the tensor itself.



Example of parametric type into a template class [4]

In order to implement a library that defines and manages a tensor, it is necessary to exploit object-oriented programming and the possibility of creating classes in C++.
Other concepts that are needed for handling a tensor are:

- **Dimensions**: they are composed by a number of integer values equal to the tensor's rank. The dimensions define the form of the tensor data structure and they are usually given in input during the definition of a new tensor.
  The dimensions have an essential role to determine the total possible number of elements that can be stored and to manage these elements.
  We have to follow the rule which requires us to maintain the data in a consecutive area.
  The dimensions are stored as fields in the tensor classes as an *array of width*.
- **Strides**: array of integers that defines the distance in the next element of this index. This array is important because it is used to pick a determined element from the tensor.
  I can transcribe an example from the delivery of the project:
  - Given a tensor of rank 3 and size (3,4,5), represented in right-major order, will have strides (20,5,1) and width (3,4,5).
  - Entry (i,j,k) will be at index (20*i+5*j+k*1) in the flat storage

- **Indexes**: they are essential to determine the position of the element to be extracted or to be picked. The indexes must be lower than the value of the dimensions inserted for the creation of the tensor object
- **Methods**: functions that belong to a class and that can be called by an object of that specific class. The methods can be accessed by an object with the dot syntax (.) if its definition into the class is specified as public.
  Methods are fundamental for handling the tensor as a data structure. The main methods required for this release are:
  - Data-access functions are used for accessing specific elements given the position given by the input indexes, for fixing an index for producing a low rank tensor that shares the same data or for producing a forward iterator
- **Iterators**: objects that can be used to loop through the whole element of the data in an ordered way. An iterator object is produced by a method of a tensor and it gives the possibility to run through the content of the elements of the corresponding tensor
- **Data**: one-dimensional support data structure used as a field within the tensor class. All items within this field are saved. Due to the demand for the possibility of sharing the same data through a game of pointers, it was necessary to define this vector of elements having the same parametric type of the class inside the smart pointer *shared_ptr*. More details will be explained on the following pages.

The introduction of the theoretical definition of a tensor makes it possible to explain more easily some specific choices made during the implementation of the library by starting as an assumption that some concepts are just explained here.

Software Implementation

Classes

We have to develop two different types of **tensors** so we created two different templated **classes**:

- ***UnknownRankedTensor***: there is no compile time information about rank or dimensions of the tensor, only type
  - The template for this class will be only the *typename* called *T* which is the name of the type to know at compile time
- ***RankedTensor***: it has rank information in its type, but no dimensional information
  - The template for this class will be the *typename T* and an *int-number* called *n*, which represents the rank to know at compile time

The declarations of the two classes will be inserted into the file ***tensor.h*** and they will be inserted into a **namespace** called ***Tensor_Library***. The namespace is a collection of entity names: its purpose is to avoid confusion and misunderstanding if many entities with similar names are needed, by providing a way to group names by categories.
A namespace is a **declarative region** that provides a **scope** to the identifiers (the names of types, functions, variables, etc) inside it. Namespaces are used to organize code into logical groups and to *prevent name collisions* that can occur especially when your code base includes multiple libraries [5].
We will use this namespace with the keyword ***"using"*** into the main in order to access directly the classes and the methods.

```
namespace Tensor_Library{
...
}
```

The **UnknownRankedTensor class** has 6 different **protected** fields:

- ***rank***, an int-attribute which keeps track of the rank's tensor
- ***sizeDimensions***, an int-vector which contains the dimensions' sizes of the tensor
- ***strides***, an int-vector attribute which contains the strides of the tensor
- ***n_total_elements***, an int-attribute which contains the value of the total elements of the tensor
- ***data***, int-vector pointer attribute which points to the vector that contains the data of tensor
- ***init_position***, an int-attribute which helps to keep track of the position in which retrieves elements or builds iterators

The reason all fields have been defined with **protected** access is that this allows direct access to subclasses that inherit those fields, which in the case of private variables was not possible to do. **RankedTensor** will be a **subclass** of **UnknownRankedTensor** and the various fields of the latter can be accessed directly from its subclass. The **reason** for this **inheritance** is the

possibility of being able to inherit the **methods** without necessarily having to override them and this allows us to save a considerable number of lines of code.

We can see that there are three modes of inheritance: [6]

- **Public Mode**: derive a subclass from a public base class
  - The public member of the base class will become public in the derived class and protected members of the base class will become protected in the derived class
- **Protected Mode**: derive a subclass from a protected base class
  - Both public members and protected members of the base class will become protected in the derived class
- **Private Mode**: derive a subclass from a private base class
  - Both public members and protected members of the base class will become Private in the derived class

```
class A {
public:
    int x;

protected:
    int y;

private:
    int z;
};
```

```
class B : public A {
    // x is public
    // y is protected
    // z is not accessible from B
};

class C : protected A {
    // x is protected
    // y is protected
    // z is not accessible from C
};

class D : private A // 'private' is default
{
    // x is private
    // y is private
    // z is not accessible from D
};
```

Example of the three inheritance modes and the members' access [6]

In our case, for this inheritance we have chosen the **first mode** in order to keep protected the fields shown before and to maintain public all the methods of the tensor object:

```
class RankedTensor : public UnknownRankedTensor<T>{
...
}
```

For the *RankedTensor* class we haven't added any more fields that aren't in its superclass.

From the **fields** of the classes we have one particular variable called data which is declared in this way:

```
shared_ptr<vector<T>> data;
```

From the documentation we can see that ***std::shared_ptr*** is a **smart pointer** that retains **shared ownership** of an object through a **pointer**. Several shared_ptr objects may own the same object.
The object is destroyed and its memory deallocated when either of the following happens: [7]

- the *last remaining shared_ptr* owning the object is *destroyed*
- the *last remaining shared_ptr* owning the object is *assigned* another pointer via *operator= or reset()*

After the initialization of a **shared_ptr object**, we can *copy* it, *pass* it as a value in function arguments, and assign it to other instances of shared_ptr. [8]
All instances point to the same object and have access to a **"control block"** that *increments* and *decrements* the ***reference count*** whenever a *new shared_ptr object* is ***added***, goes *out of scope*, or is *reset*. When the reference count reaches ***zero***, the control block *discards* the *memory* resource and *itself*. [8]

This **pointer field** called *"data"* will **point** to an object whose type is an **int-array** containing all the **data**'s tensor.
Many **methods** belonging to the tensor classes that will return new tensors in **two different versions**:

- *returning* a *new tensor object* that contains a *copy of the tensor data* on which this method has been applied or a copy of a portion of that data vector
- *returning* a *new tensor object* that contains a data field that *points to the same data* of the tensor on which this method has been applied or to a portion of it

It was necessary to use this **smart pointer** precisely for the **second** of the two **versions** mentioned because the new tensor produced **points** to the **data** of the tensor on which the method was applied.

Constructors

The two classes have more definitions of constructors because we can have different inputs.
We firstly consider the ***UnknownRankedTensor class*** with the following **constructor**:

```
template <typename T>
UnknownRankedTensor<T>::UnknownRankedTensor(vector<int> args){
    ...

    for(int i = rank-1; i >= 0; i--){
        if(args[i] <= 0)
            throw invalid_argument("A dimensional space cannot be zero or less");
        else{
            strides[i] = n_total_elements;
            n_total_elements *= args[i];
            sizeDimensions[i] = args[i];
        }
    }
}
```

The **input** is a vector of integers which contains all the size for all dimensionality of the tensor. With this vector we ***extract the rank*** of the tensor, which must be greater than zero, otherwise it throws an exception.
From the portion of code seen before we can see the **computation** of the variable which represent the value to the *total* ***number of elements*** (initially set to one) and the ***two int-vectors*** corresponding to the ***strides*** and the ***size dimensions***:

- The *vector sizeDimensions* corresponds to the input vector and each element is copied
- The *total number of element* has a progressive product of each element of the input
- The *vector strides* needs the the progressive product of each element made to compute the number of elements in the tensor and, for each step of iteration, it saves the values

The initial position value used to compute the index for extracting an element for the tensor is initially set to one.

This is one of the **three overloading constructors** of the same class:

- It applies the same instructions of the previous constructor by converting the series of input integers required into a vector of integers

```
template <typename T>
template <typename... ints>
UnknownRankedTensor<T>::UnknownRankedTensor(ints...sizes) :
UnknownRankedTensor<T>::UnknownRankedTensor(vector<int>({sizes...})){ }
```

There are other two overloading constructors corresponding to the ***copy-constructor*** and the ***move-constructor*** inserted for the **rule of Three/Five/Zero**. More explanations for their usage and their existence are inserted in the next chapter.

This is the **constructor** of the ***RankedTensor class***:

```
template <typename T, int n>
RankedTensor<T, n>::RankedTensor(vector<int> args) :
UnknownRankedTensor<T>::UnknownRankedTensor(args){

    int size = args.size();

    if(n != size)
        throw invalid_argument("The number of space dimensions inserted are not
equal to the number of rank");

    this->rank = n;
}
```

We know that ***RankedTensor*** is a **subclass** of the UnknownRankedTensor class. This constructor applies the **instructions** of the constructor's **superclass** which takes in input the vector of sizes' indexes.
Subsequently it was checked whether the **value of rank**, put in input and known at compile time as **n**, is **equal** to the length of the vector of the sizes' dimensionality and finally the value of *n* was saved in the *rank field* defined in its superclass.

Also in this class there are the **same three overloading constructors** defined in its superclass of which no further explanation is needed since the same reasoning described before.

Rule of Three/Five/Zero

The **Rule of five** is a programming concept introduced with **C++11**.
It originates from the Rule of Three, where the introduction of **Move Semantics** in C++11, caused the **Rule Of Three** to expand and become the **Rule Of Five** [9].

The **Rule of Three** states, that if a Class explicitly defines any of the following special member functions, then it is required to define the others: [9] [10]

- *Destructor*
- *Copy Constructor*
- *Copy Assignment Constructor*

In short, if a class **explicitly defines** the *Destructor*, then it **should explicitly define** the *Copy Constructor* and the *Copy Assignment Constructor* too.
An explicitly defined function is basically when the user himself defines the function.
There are already default implementations of these member functions but they perform in a very basic way [9].
In our classes of tensors for example we have to define the *Destructor* because the tensor deals with the dynamic memory for its data and the indexes and it's important to eventually free that memory. In order to follow this programming concept we have also to explicitly define the *Copy Constructor* and the *Copy Assignment Constructor*.

Since we decided to implement the tensor library with a version of C++ higher than the eleventh, we have also to follow the **Rule of Five**.
Basically, if a class explicitly defines any of the following special member functions, then it is required to define the others: [9] [10]

- *Destructor*
- *Copy Constructor*
- *Copy Assignment Operator*
- *Move Constructor*
- *Move Assignment Operator*

The *Move Constructor* and *Move Assignment Constructor* are inserted in order to deal with the inefficiency introduced by Copy Constructors when dealing with memory.
Let's imagine a situation where we have made the follow code example: [9]

```
int main() {
     ClassA objA = ClassA( String("Hello World");
}
```

```
String(const String& str) {   // Copy Constructor
    cout << "Copy Constructor Called\n";
    _size = str._size;
    _data = new char[_size];
    memcpy(_data, str._data, _size);
}
```

For the nature of the Copy Constructor, we will end up with two instances of the same memory. The string "Hello World" will first be allocated in the main() function, and then copied into ClassA. When ClassA copies it, it will create new memory for the String:

This is a problem because memory is allocated twice and also deallocated twice. For this reason we need Move Constructors, which “move” the memory, instead of copying it.

The rule of zero says that the classes which have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership (which follows from the Single Responsibility Principle). Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators[1].

The **Rule of Zero** explicits that the classes which have custom destructors, copy/move constructors or copy/move assignment operators should deal exclusively with ownership. Other classes should not have custom destructors, copy/move constructors or copy/move assignment operators [10].
When a base **class** is intended for **polymorphic** use, its ***destructor*** may have to be declared **public** and **virtual**. This blocks implicit moves (and deprecates implicit copies), and so the **special member functions** have to be declared as **defaulted**. However, this makes the class prone to slicing, which is why polymorphic classes often define copy as deleted.
This rule also appears in the *C++ Core Guidelines as C.20:* “If you can avoid defining default operations, do” [10].

The set of these three rules composes a single concept called **Rule of Three/Five/Zero** and this is what we followed for the *UnknownRankedTensor* class and its subclass *RankedTensor*.

In our class we have defined all of these special member functions from the rule of five on the file ***tensor.h*** with the **default** keyword. It means that we want explicitly to use the compiler-generated version of that function, so we don't need to specify a body.
The keyword **delete** instead specifies that we don't want the compiler to generate that function automatically.
The keyword **virtual** for a function says that it’s declared within a base class and is re-defined (overridden) by a derived class. The ***destructor*** of the superclass must be ***virtual*** in order to correct the situation in which the deleting of a derived class object using a pointer of base class type that has a non-virtual destructor would result in undefined behavior [11].

This is the example of application of the *Rule of Three/Five/Zero* for *UnknownRankedTensor*:

```
virtual ~UnknownRankedTensor() = default;
UnknownRankedTensor(UnknownRankedTensor<T> &tensor) = default;
UnknownRankedTensor& operator=(const UnknownRankedTensor& tensor) = default;
UnknownRankedTensor(UnknownRankedTensor<T> &&tensor) = default;
UnknownRankedTensor& operator=(UnknownRankedTensor&& tensor) = default;
```

Methods

Almost all the **methods** are public and they are all declared into the tensor.h file. Their implementations are written in *UnknownRankedTensor.cpp*, *RankedTensor.cpp* and *Utils.cpp*. We have decided to use different files for the methods' implementation for shaping the structure of the project based on the classes in which they belong or their functionality.

In this report we have divided the methods into **four different categories**:

- ***Getter and Setter***: methods used to return protected or private fields
- ***Required methods***: they are requested from the delivery of the project, so they must be implemented
- ***Methods for testing***: they are implemented for showing that the implementation of the required methods are correct. Their implementations are contained in a file into a folder called *Utils*.
- ***Methods for the Iterator and the Einstein Formalism***: they produce objects such as *TensorIterators* and *TensorWithIndexes* for, respectively, iterating tensors and applying operations

Getters and Setters

The **getter** and **setters** are simple methods that are present in many classes which give the opportunity to access or change a specific private or protected field.

The *getter* defined for both tensor classes are:

- *getRank*: it returns the rank of the tensor
- *getSizeDimensions*: it returns the int-vector that contains the dimensions of the tensor
- *getStrides*: it returns the vector of integer numbers corresponding to the tensor's stride
- *getData*: it returns the pointer to the vector of data
- *getInitPosition*: it gives the value of initPosition, useful for the computation of the value to access
- *get_n_total_elements*: it gives the number of elements stored into the tensor

The *setter* defined for both tensor classes are:

- *setStrides*: it allows to replaces the vector of tensor's strides with the input vector
- setData: it allows to replace the previous data pointer with an input pointer of data
- *setInitPosition*: it allows to change the value of the initial position for the competition of the index of the value to access

All of these methods in the class *RankedTensor* are inherited from the class *UnknownRankedTensor* by exploiting the keyword ***using***, without giving an overridden implementation. These methods are useful for obtaining the access of determinate fields on the required methods because they can modify the structure of the belonging tensor or they can produce a new tensor.

Required methods

In this subsection we will focus on all the required methods present in the two tensor classes.
The **public methods** implemented are:

- ***get***: it returns an element of the tensor given the initial position and the indexes in input
- ***set***: it sets the input value of an element given the initial position and the input indexes
- ***fix***: it slices the tensor by fixing a specifying space. It returns a tensor with a rank reduced by one and with the data that points to the original tensor in which the method is applied
- ***fix_copy***: it slices the tensor by fixing a specifying space. In this case it returns a new tensor with a rank reduced by one and the data field points to a completely new vector of data in which the interested elements are copied from the original tensor
- ***flattening***: it applies the flattening to a tensor. The method will return a tensor with rank and the data will point to the original tensor one.
- ***flattening_copy***: it applies the flattening to a tensor. The method will return a tensor with rank and the data doesn't point to the original data tensor.
- ***window***: By giving in input the vector of initial indexes and the vector of end indexes it will produce a sub-window of a given tensor in which the interested portion of data will points to the portion of data the original tensor
- ***window_copy***: By giving in input the vector of initial indexes and the vector of end indexes it will produce a sub-window of a given tensor in which the interested portion of data is copied from the data of the original tensor without pointing to it

The method ***get*** is important because it will be used to retrieve an element by taking into account the field *initial_position* which can be modified by some other methods such as *window* or *fix*. It will be used for printing the elements in methods like *printData* because it computes the position of the interested element to print for the tensor.
This method will be overloaded in order to give in input a series of indexes instead of a vector.
This is the **computation** of the **index's vector** in the method *get*:

- It applies the sum in each iteration of the cycle in which is computed the product between the strides element and the tensorIndexes in input. The index value starts from the initial position stored into the tensor (initially set to zero from the constructor)
- The ***tensor*** with ***rank equal to zero*** requires an empty input tensorIndexes or a vector with only one single element setted to zero because it contains one element

```
template <typename T>
T UnknownRankedTensor<T>::get(vector<int> tensorIndexes){
    int index = init_position;

    ...

    for(int i=0; i<rank; i++)
        index += strides[i] * tensorIndexes[i];

    return data->at(index);
}
```

We have defined some **operators** such as *operator()* for retrieving an element without specifying the method *get* and applying it only by giving the indexes of the element to retrieve.
We have implemented another operator for the application of the method *set* by giving only the indices of the element to override.

The difference between the methods with the word ***copy*** into the name and the ***others*** is that the methods with ***copy*** produces tensors in which the data is **copied** from the original tensor but the **pointer** of the data field will points to a completely **new vector**, while in the methods ***without copy*** the data field of the new tensor will **point** to the **data** of the **original tensor** in which the method is applied.
We show a concrete **example** from ***flattening*** (which is the simplest method):

```cpp
template <typename T>
UnknownRankedTensor<T> UnknownRankedTensor<T>::flattening(){
    // It doesn't matter if the rank of the original tensor
    UnknownRankedTensor<T> newTensor(n_total_elements);
    newTensor.setData(data);
    return newTensor;
}
```

```cpp
template <typename T>
UnknownRankedTensor<T> UnknownRankedTensor<T>::flattening_copy(){
    UnknownRankedTensor<T> newTensor(n_total_elements);

    // Creating a new vector with the same elements of the original tensor
    shared_ptr<vector<T>> newData = make_shared<vector<T>>(n_total_elements);

    int index = 0;

    // Inserting the elements of the new tensors and managing the case if the tensor is a trace
    if(getRank() == 0)
        newData->at(index) = get();
    else{
        TensorIterator<T> it = getIterator();

        while(it.hasNext()) {
            newData->at(index) = it.next();
            index++;
        }
    }

    // Setting into a new tensor the new vector containing data
    newTensor.setData(newData);
    return newTensor;
}
```

The class *RankedTensor* inherits the methods get and set through the keyword using, but we preferred to **override** all the methods *fix*, *flattening* and *window* (and their *copy* versions) because we have chosen to produce as **output** an object with *RankedTensor*, in which the **rank** is known at **compile-time**, and this is not possible if the methods were completely inherited.

Methods for testing

These methods are divided into the print methods and the methods used for filling a tensor with random arithmetic data.

The ***print methods*** are two:

- *printData*: it uses an iterator to print all the data of the tensor
- *printTensor*: it prints all the main fields that characterized the tensor:
    - sizeDimensions
    - strides
    - Number of the total tensor's element
    - The full content of the data stored into the tensor by using as auxiliary function the printData method

We have used two functions in order to allow us to completely fill the tensor with random elements of the parametric *type T*.

The first method called ***randomNumber*** is **protected** because it's used only as an auxiliary function and it's **static** because we don't need to apply it to a tensor object.
We have setted a *32-bit pseudo-random number generator* using the *Mersenne algorithm* (which we didn't study) and we apply a uniform distribution with that generator.
We checked at compile-time whether the parametric type is *integral* or *floating point*:

- An ***integral type*** is a set of types (such as int, char, unsigned int, …) in which the elements are integers numbers
- A ***floating point type*** is a set of types (such as float, double, unsigned float, …) in which the elements are floating numbers
- If the *type T* doesn't correspond to a type which is floating point or integral, it throws an exception because this function works only for *arithmetic types*

The **public** method ***insertRandomData*** will insert all the random numbers of type T produced semi-randomically into the field data of the tensor. It checks if the type T is *arithmetic* for the correct execution of this method.
For each *checking* the *type T* we have to use the keyword ***constexpr*** since it's introduced for the version *C++11*: this specifier declares that it is possible to evaluate the value of the function or variable at compile time. This is important because the construction of the object *distr* which can be *uniform_int_distribution<T>* or *uniform_real_distribution<T>* must be known at compile-time, otherwise we get an unexpected and unhandled *exception* from the compiler.

Methods for the Iterator and the tensor operations

We also have two versions of the method ***getIterator*** which returns an object of type *TensorIterator<T>* which will iterate a portion or the full content of the data's tensor.
The *version* which iterates to the *full content* of the data's tensor has no input elements, while the *overloaded second version* which iterates to a portion of data must have as input a vector of fixed indices of size rank-1 (because one space must be free) and the space which is not fixed.
This *second version* of this method allows us to see content along any one index, keeping the other indices fixed.

Another important method is the ***operator*** with ***round brackets*** which takes in **input** a ***vector of Index objects*** in which a tensor can produce an object called *TensorWithIndexes*. We will focus on this in the section reserved to the Einstein Formalism and the operations between tensors.

All of these methods are completely **inherited** from the class *RankedTensor* with ***using***.

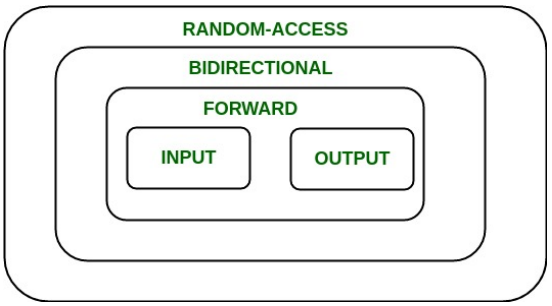
Iterators

Theoretical definition

An **iterator** (or **cursor**) is an object that lets us visit sequentially all the elements contained in another object, typically a container of elements (such as lists, vectors, matrices or tensors), without worrying about the details of the implementation of the container to iterate.
The iterators ensure the **access** of the **element** actually pointed and the updating of the **pointer to the next element** of the container. In this way we have introduced a specific pattern.
In *object-oriented programming*, the **iterator pattern** is a design pattern in which an iterator is used to traverse a container and access the container's elements [12].

The **iterators** can be be distinguished in [13]:

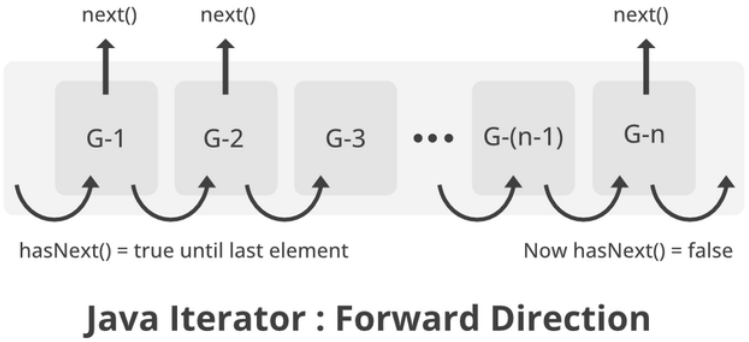
- ***Input iterators***: weakest of all the iterators and have very limited functionality. They can only be used in single-pass algorithms, i.e., those algorithms which process the container sequentially, such that no element is accessed more than once
- ***Output iterators***: very limited in their functionality and can only be used in single-pass algorithm, but not for accessing elements, but for being assigned elements
- ***Forward iterators***: higher in the hierarchy than input and output iterators. They contain all the features present in these two iterators, but they also can only move in a forward direction and that too one step at a time
- ***Bidirectional iterators***: since they can move in both the directions, they have all the features of forward iterators along with the fact that they overcome the drawback of forward iterators
- ***Random Access iterators***: most powerful iterators. They are not limited to moving sequentially because they can randomly access any element inside the container. They are the ones whose functionality are same as pointers



Graphic that shows the powerful classification of functionality of the iterators [13]

The iterators can let us visit **all the elements** contained into an object **or** only on a **portion** of elements from the container.
In our specific case, for the **tensor** we have to provide an iterator class that must provide *forward iterators* to the *full content* of the tensor or to the *content along any one index*, keeping the other indices fixed.

For our project we have built the iterator by inspiring on the concept of Java's iterators and we have inserted the two methods *next()* and *hasNext()* which allow us to iterate the elements of the tensor. In order to compute the index for retrieving the element from the data structure we have used the method *get()*, already implemented for the tensors' classes.



Example of Java's iterators [14]

Usually the iterators in C++ are builded by exploiting the knowledge of the **Standard Template Library** (**STL**), a set of C++ template classes to provide common programming data structures and functions such as lists, stacks, arrays, etc. It is a library of container classes, algorithms, and iterators [15].

We have **preferred** to exploit **Java's standard concept** of **iterators** because it's more familiar and certainly simpler than C++ STL's approach.
Surely the iterator class produced with the developed Java's approach will not miss any of the project's requests and will be able to iterate easily on all the elements of the tensor or on a specific range specified by the input parameters in the initialization phase.

Software implementation

Class

We have built only one parametric class in order to represent the tensor iterators.
The declaration of this class is put into a file called ***iterators.h*** stored into a folder called *"Iterators"*. Into the same folder there will be present the file ***iterators.cpp*** with the implementation of the constructors and the methods of the class.

This **declaration** of the two tensor's classes into the namespace *Tensor_Library* is **necessary** because for the iteration's class we need to use their classes and the corresponding file with their declarations isn't imported yet with the declarations of the two classes.
In case we import that file we would run into a conflictual error and in case we reverse the order of importation we have to insert the declaration of the iterator's class into its namespace *Iterators* into the tensor.h file.

```
namespace Tensor_Library{
    template <class T, int n>
    class RankedTensor;

    template <class T>
    class UnknownRankedTensor;
}
```

We have decided to insert the class related to the Iterator into a **namespace** called *Iterators*.

```
namespace Iterators{
...
}
```

We have opted for using only one Iterator class for the two types of tensors by exploiting the fact that the RankedTensor class is a subclass of UnknownRankedTensor class.
This class, called TensorIterator, is templated with a parametric type T which corresponds to the type of the associated tensor.

```
template <typename T>
class TensorIterator {
    ...
}
```

This class contains **seven different private fields**:

- The pointer to the associating tensor
- The vector indexes which contains the coordinates of the current position
- The vector indexes which contains the coordinates of the last element to iterate
- Two vectors used for iterate only the content along any one index, keeping the other indices fixed

- Boolean which specific if we have to iterate all the content of the tensor or not

This class contains only **three public methods**:

- ***hasNext()***: it returns a boolean which says whether is present a successive element
- ***next()***: it returns the current element and it jumps to the coordinates of the next one
- ***getIndexes()***: returns the current coordinates position of the iterator, it allows us to acceeds to the private field which corresponds to the vector ***indexes***

This iterator works for both the tensors because it uses the same common properties and it saves as a field a pointer to the associated tensor of type *UnknownRankedTensor*. Since that class object is a **superclass** of *RankedTensor*, I can store into that field an object of that type class by applying an automatic **subsumption**, which lets us build only one iterator for both the tensors.
Obviously this common iterator class must have a **rank** which is **not known** at **compile-time** and it's extracted by the associating tensor from the constructor.

Constructors

This is the **most common constructor** built for the TensorIterator type:

```
template <typename T>
TensorIterator<T>::TensorIterator(UnknownRankedTensor<T> &tensorInput) :
tensor(&tensorInput){

    if(tensorInput.getRank() <= 0)
       throw invalid_argument("The rank for the production of an iterator
must be greater than zero");

    n = tensor->getRank();
    indexes = vector<int>(n);
    endIndexes = vector<int>(n);
    this->isIteratorAllContent = true;

    for (int i = 0; i < n; i++) {
        indexes[i] = 0;
        endIndexes[i] = tensor->sizeDimensions[i] - 1;
    }
}
```

The **only input** of the constructor will be the **tensor** in which we have to iterate all the data.
It initializes the coordinates indexes to zeros, i.e. the first element of the tensor, and the coordinates of the last element of the tensor.

The iterator produced by this constructor will iterate to all the data's tensor. This constructor is related to the tensor's method getIterator() which has no input parameters and it passes itself to the iterator's constructor:

```
template <typename T>
TensorIterator<T> UnknownRankedTensor<T>::getIterator(){
    TensorIterator<T> iterator(*this);
    return iterator;
}
```

This overloading constructor differs to the previous one for the two inputs in addition to the tensor:

- ***excludingSpace***: it corresponds only space which is not fixed
- ***inputIndexes***: it corresponds to a vector of indexes for each dimension (except one) of the tensor that will be taken fixed when the iterator will iterate the portion of data requested

```
template <typename T>
TensorIterator<T>::TensorIterator(UnknownRankedTensor<T> &tensorInput, int
excludingSpace, vector<int> inputIndexes) : tensor(&tensorInput){

    ...

    int i;
    for (i=0; i<=n-1; i++) {
            this->tensorSpaces.push_back(i);
        }

    this->tensorSpaces.erase(tensorSpaces.begin() + excludingSpace);
    this->tensorIndexes.insert(tensorIndexes.begin(),
              std::begin(inputIndexes), std::end(inputIndexes));
    this->isIteratorAllContent = false;

    bool flag = false;
    int sup = 0;

    for (int i = 0; i < n; i++) {
        if(i == excludingSpace){
            indexes[i] = 0;
            endIndexes[i] = tensor->sizeDimensions[i] - 1;
            flag = true;
       } else {
            if (flag) sup = 1;
            indexes[i] = inputIndexes[i- sup] ;
            endIndexes[i] = inputIndexes[i- sup];
       }
    }
}
```

In this case the initial indexes will not necessarily be composed by zeros and the final indexes will not necessarily be composed by all the dimensions minus one because this iterator is projected to iterate in a **specific portion** of the **data's tensor**.

This overloading constructor is related to the overloaded version of tensor's method ***getIterator()*** which has two input parameters corresponding to the space which is not keeped fixed and to the vector of integer which contains all the fixed indexes for each dimensional space, except the non-fixed one:

```
template <typename T>
TensorIterator<T> UnknownRankedTensor<T>::getIterator(int excludingSpace,
vector<int> inputIndexes){
    TensorIterator<T> iterator(*this, excludingSpace, inputIndexes);
```

```
    return iterator;
}
```

There is also another similar method with the same name which allows us to run the same previous method, but the indexes can be expressed as a list of int-arguments.

Methods

For the class ***TensorIterator*** we have implemented only **three methods**. We have decided for the rule of three/five/zero to explicitly define none of the five special member functions.
Two of the three methods' iterators can be grouped into the ***required methods*** because they are necessary for the correct implementation of the main functionality of the class and the other method is a ***getter*** function.

Getters and Setters

The only **getter** method is called **getIndexes** and it simply returns a vector of integers called *indexes* which corresponds to the set of the current indexes reached during the iteration.
It is necessary because the class *TensorIterator* cannot be a friend class with the class *RankedTensor* because of the different templates, so an object tensor of that class cannot access the private field *indexes*.
Instead of changing the private fields from ***private*** to ***protected***, we have decided to insert this methods which will be used during the implementation of the methods *fix_copy* and *window_copy* for the class *RankedTensor*.
In the *TensorIterator* class it was **not necessary** to introduce any **setter** method and no other **getter** method.

Required methods

There are only **two required methods** inspired from Java's implementation of the iterators: ***hasNext*** and ***next***.

The **hasNext method** has an important role because it checks whether there is a successive element to iterate or not.
The first and the last *if statements*, in combination with the next method's instructions, is useful to allow the array index to access the last element even though there are no new elements after it.

```
template <typename T>
bool TensorIterator<T>::hasNext(){
    for(int i = 0; i < n; i++){
        if(indexes[i] < endIndexes[i]) return true;
        else if(indexes[i] > endIndexes[i]) return false;
    }
    if(indexes[0] == endIndexes[0])
        return true;
    return false;
}
```

The **next method** allows us to return the current element and to jump into the next element.
The difficulty in implementing this method is to increase the indices from a vector composed of several elements corresponding to each index of the tensor.
Note the fact that the return of the element of *parametric type T* of the tensor given the indices is performed with the help of the ***get method*** performed on the tensor associated with the iterator.
In this way the calculation of the *right index* of the vector containing the data is always performed via the *get method*.

```
template <typename T>
T TensorIterator<T>::next(){
    int idx = n-1;
    bool check = false;
    int checkLast = indexes[0];
    int sup = 1;

    vector<int> vectIndexes(begin(indexes), end(indexes));
    T elem = tensor->get(vectIndexes);

    while(!check && idx >= 0){
        if(indexes[idx] == endIndexes[idx]){
            if(isIteratorAllContent || idx != tensorSpaces[idx-sup]) {
                indexes[idx] = 0;
                sup = 0;
            } else {
                indexes[idx] = tensorIndexes[idx-sup];
            }
            idx--;
        }
        else{
            indexes[idx]++;
            check = true;
        }
    }

    if(!check)
        indexes[0] = checkLast + 1;

    return elem;
}
```

Operations between tensors

In this section, two operations required for tensor design will be addressed as topics: the algebraic sum and the product.

Introduction of Useful Classes

Before going into the discussion of operations, we decided first to talk about **two** of the **classes** introduced to be able to perform them.

The first class introduced is called **Index**. It looks a lot like a wrapper class, however it is used to be able to distinguish an index whose only field, which will be assigned by the constructor with the integer input, is a value indicating the space to which it refers.
The only **public method** is called ***getSpace***, a getter that returns the value of space's field.

The other class introduced is called **tensorWithIndexes**. The constructor has in input:

- An ***associated tensor*** of type *UnknownRankedTensor* or its subclass in which will be applied operations with another tensor associated to another object of this new class
- A ***vector*** which contains the same number of *Index* objects of the rank's associated tensor. We have added the rule in which we can't insert in this vector two Index objects with the same space's value

The other constructor allows us to insert the Index objects without putting them into a vector. It applies the same instructions of the main constructor.

The **only two public methods** of this class are two getters:

- *getTensor*: which returns a pointer to the associated tensor
- *getVectorIndexObjects*: which returns the vector that contains the Index objects

We have defined another class, called **MultiplierTensor**, which will be used for the application of the product between tensors. It is basically an "accumulator" of the factors and the data which will be used for the operation.
This class is used for postponing the product because the *operator* * produces a MultiplierTensor object that saves the indexes and the parameters in order to be careful to not lose some important information. Eventually the product will be applied from a public method of that class that exploits the saved attributes.

The **operators +** and * are declared into the file *operators.h* and implemented into the file *operators.cpp*: the true product will be postponed into a method contained by the MultiplierTensor while it's not necessary to be postponed, so it will be applied into the operator.
The two operators and the Multiplier object can be invoked or created only with ***arithmetic types*** not corresponding to *char*, *unsigned char*, *signed char* or *bool*.

Algebraic Sum

This section will deal with the implementation of the **algebraic sum** between tensors.
The **operator +** method allows us to apply the ***algebraic sum*** between two *tensorWithIndexes* objects.
We can apply the algebraic sum only to tensors with *indexes* inserted in the same order with the same spaces and the *sizeDimensions* of each corresponding index must be equal and in the same order. This operator can be applied in a recursive way to different *tensorWithIndexes* objects.
The **result** can be seen to the object produced by the result of this operation, by extracting the associated tensor and printing the elements contained into that data structure.

Software Implementation

This is the implementation of the **operator +** in the file *operators.cpp* after the declaration into the *.h* file in the same folder.

```
template <typename T>
TensorWithIndexes<T> operator +(TensorWithIndexes<T> tensorWithIndexes1,
TensorWithIndexes<T> tensorWithIndexes2) {
    ...
}
```

The first part of the sum operator deals with the management of possible **exceptions** in the following cases:

- The two tensors to be added don't have the same *ranks*
- The two *tensorWithIndexes* objects to be added don't have the same spaces' value for each corresponding Index object in the same order
- The two tensors to be added don't have the same *sizeDimensions* elements' value, in the same order

The second part of the instructions deals with the **algebraic sum** between the elements of the two tensors associated with the two *tensorWithIndexes* objects.
We have produced a ***new data vector*** which will contain every element which is the result of the sum between the two addends from the two tensors in the same positions.
In order to do that we have ***built two iterators*** for the two tensors and, with the usage of a while cycle, we extract the two elements (which will be with the *same type T*), we sum them and we insert them into this new vector.
Eventually, we apply the method *getTensor* and *setData* in order to insert the new vector of data to the associated tensor of the tensorWithIndexes1 parameter.
Eventually the **result**, which is stored into the first TensorWithIndexes parameter, will be returned.
On the next page, we report the *second part* of the instructions of the *operator +*:

```
template <typename T>
TensorWithIndexes<T> operator +(TensorWithIndexes<T> tensorWithIndexes1,
TensorWithIndexes<T> tensorWithIndexes2) {

    ...

    int n_total_elements =
tensorWithIndexes1.getTensor().get_n_total_elements();
    shared_ptr<vector<T>> newData =
make_shared<vector<T>>(n_total_elements);

    TensorIterator<T> it1 = tensorWithIndexes1.getTensor().getIterator();
    TensorIterator<T> it2 = tensorWithIndexes2.getTensor().getIterator();
    int index = 0;

    while(it1.hasNext() && it2.hasNext()){
        T elem1 = it1.next();
        T elem2 = it2.next();
        newData->at(index) = elem1 + elem2;
        index++;
    }

    tensorWithIndexes1.getTensor().setData(newData);

    return tensorWithIndexes1;
}
```

Product

This section will deal with the implementation of the **product between tensors**. The correct view to be able to apply a product between tensors is to continue using the *Index* and *tensorWithIndexes* classes and to build the **operator *** for this last class.
In order to obtain the correct result, one must pay attention to ***Einstein's Formalism*** and to the possible ***recursive use*** of the same * operator on multiple *tensorWithIndexes* objects.

Einstein's Formalism

The **Einstein notation** (or **Einstein Formalism**) is a notational convention that implies summation over a set of indexed terms in a formula, thus achieving brevity.
The main concept is that, when an index variable appears twice in a single term and it's not otherwise defined, it implies summation of that term over all the values of the index.
So where the indices can range over the set {1, 2, 3}: [16]

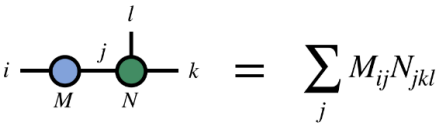
$$y = \sum_{i=1}^{3} c_i x^i = c_1 x + c_2 x + c_3 x$$

is simplified by the convention to:

$$y = c_i x^i$$

The application of Einstein's formalism for this project consists in applying the product between **tensors**, so it's necessary to use the concept of ***Tensor Diagram Notation***, a simple, yet powerful graphical notation for networks of contracted tensors [17].
This concept lets us simplify the application of the Einstein Formalism for the tensor products.

A tensor contraction of the form $\sum_j M_{ij} N_{jkl}$ can be notated: [17]



$$= \sum_j M_{ij} N_{jkl}$$

The ***Tensor Diagram*** concept has **two rules**:

- Tensors are notated by shapes (usually filled or shaded), and tensor indices are notated by lines emanating from these shapes
- Connecting two index lines implies a contraction, or summation over the connected indices

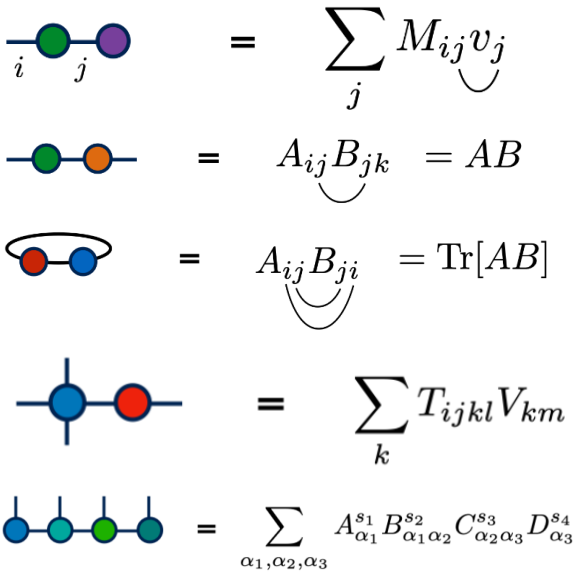
These are examples of common low-order tensors: [17]

vector v_j

matrix M_{ij}

3-index tensor T_{ijk}

These are some examples of matrices and tensors contractions: [17]



We can see that the products between tensors of different space dimensions cause the contractions of the tensor result. In the third example the final result of the matrix product (tensors of rank 2) with indexes i and j gives as result a trace of AB.
The ***trace*** of a square matrix A, denoted ***tr(A)***, corresponds to the sum of elements on the main diagonal (from the upper left to the lower right) of A. The trace is only defined for a square matrix (n × n). [18]
In this case the trace of AB for the indexes i and j corresponds to:

$$tr(AB) = \sum_{i,j} A_{ij} B_{ij}$$

An example of product is shown into the delivery of the project: the expression $a_{ijk}\, b_j$ represents a tensor c with rank 2, indexed by i and k, such that:

$$c_{ik} = \sum_j a_{ijk} b_j$$

The **product** will work with both the classes implemented for tensors:

- *UnknownRankedTensor*: tensors with no compile time information about rank or dimensions of the tensor, only type
- *RankedTensor*: tensors with rank information in its type, but no dimensional information

Software Implementation

The **operator *** is defined in the *operators.h* file and implemented in the *operators.cpp* file.
This operator has been defined in **two different methods** that both return an object from the *MultiplierTensor* class that they take as input respectively:

- *two TensorWithIndexes* objects to multiply
- a *MultiplierTensor object* and a *TensorWithIndexes object*

The **MultiplierTensor class** has been defined as a factor accumulator to be able to perform multi-factor multiplications without losing fundamental information such as the common and uncommon indices of each of the tensors with which to perform the product.
Therefore the operator * does not directly execute the product, but it saves all the necessary information for the application of the multiplications inside an object of class MultiplierTensor.
The two operator definitions are required respectively for a two-factor product and a more than two-factor product.
We won't show the implementation of the operators * in this report because in that part there isn't the true product, but only the phase of preparation of the data in order to apply the product later.

An ***example*** of application of the product is the following:

```
MultiplierTensor<int> mtest = a * b * c;
TensorWithIndexes<int> ris = mtest.applyProduct();
ris.getTensor().printData();
```

We can say that a, b and c are tensorWithIndexes objects and they produce a MultiplierTensor object because:

- a and b applies the operator * with two tensorWithIndexes objects and it produces a MultiplierTensor object
- The MultiplierTensorObject produced by the previous method and the TensorWithIndexes object c are the input arguments and they produce the final MultiplierTensor object

Eventually, the real product is applied after invoking the ***applyProduct method*** to the MultiplierTensor object produced by the *operator* *, in which are saved all the useful attributes for the application of the product.

Now we can talk about the structure of the **class MultiplierTensor**. It has ***two constructors***:

- The first one takes in input two TensorsWithIndexes objects (the two factors), the two maps of equal and different indexes between the two tensors and the vector containing the Index objects of different indexes. It also checks whether the type is *arithmetic*, but not equal to *char*, *unsigned char*, *signed char* and *bool*.

```
template <typename T>
MultiplierTensor<T>::MultiplierTensor(TensorWithIndexes<T> fact1,
TensorWithIndexes<T> fact2, map<int,int> mapOfDifferentIndexesInput,
map<int, int> mapOfEqualIndexesInput, vector<Index>
vectorDifferentIndexesInput){

    if constexpr(!is_arithmetic_v<T>) throw invalid_argument("The type
must be arithmetic!");
    if constexpr(is_same_v<char, T> || is_same_v<signed char, T> ||
is_same_v<unsigned char, T>) throw invalid_argument("The type shouldn't be
char because there is an high overflow risk");
  if constexpr(is_same_v<bool, T>) throw invalid_argument("The type
shouldn't be bool because there is an high overflow risk");

    mapOfDifferentIndexes = map<int,int> (mapOfDifferentIndexesInput);
    mapOfEqualIndexes = map<int, int> (mapOfEqualIndexesInput);
    vectorDifferentIndexes = vector<Index>(vectorDifferentIndexesInput);

    factors = vector<TensorWithIndexes<T>>();
    factors.push_back(fact1);
    factors.push_back(fact2);

    prod_result = UnknownRankedTensor<T>();
}
```

- The second constructor takes in input another MultiplierTensor and it produces another MultiplierTensor by copying the values of attributes of the input argument

```
template <typename T>
MultiplierTensor<T>::MultiplierTensor(MultiplierTensor<T> &mt){
    mapOfDifferentIndexes = map<int,int> (mt.mapOfDifferentIndexes);
    mapOfEqualIndexes = map<int, int> (mt.mapOfEqualIndexes);
    vectorDifferentIndexes = vector<Index> (mt.vectorDifferentIndexes);
    factors = vector<TensorWithIndexes<T>>(mt.factors);

    prod_result = UnknownRankedTensor<T>(mt.prod_result);
}
```

The two constructor are implemented in synergy with the operator * in order to obtain all the useful attributes into the object of this class.

The class MultiplierTensor is composed by the following ***private attributes***:

- *mapOfDifferentIndexes*, a map of two integers which corresponds respectively with the identifier space of the index as key and the total dimension as value. This map contains all the indexes which are not common between all the factors and they will compose the set of indexes of the tensor produced by the product
- *mapOfEqualIndexes*, a map of two integers which corresponds respectively with the identifier space of the index as key and the total dimension as value. This map contains all the indexes which are common between all the factors and they will compose the set of indexes in which will be applied the Einstein Notation
- *vectorDifferentIndexes*, vector which contains a set of Index objects that is useful for producing a TensorWithIndexes from an UnknownRanked tensor object
- *factors*, a vector containing the TensorWithIndexes objects that are the factors of the product
- *prod_result*, UnknownRankedTensor object which will save the result obtained after invoking the applyProduct method

There are also a lot of ***getters*** and ***setters*** useful to get and update all the attributes of the object such as: *getFactors*, *getMapOfDifferentIndexes*, *getMapOfEqualIndexes*, *getVectorDifferentIndexes*, *setMapOfDifferentIndexes*, *setMapOfEqualIndexes*, *setFactors* and *setVectorDifferentIndexes*.

Now we can focus about the **applyProduct method**: it exploits two private methods: *prod* and *recursiveProduct*. The method prod is an auxiliary function of recursiveProduct, which is an auxiliary function of the applyProduct method itself.
The public method ***applyProduct*** is implemented in the following way:

```
template <typename T>
TensorWithIndexes<T> MultiplierTensor<T>::applyProduct(){

    vector<TensorWithIndexes<T>> factors = getFactors();
    map<int, int> commonIndexes = getMapOfEqualIndexes();
    map<int, int> nonCommonIndexes = getMapOfDifferentIndexes();

    vector<int> sizeDimensionsDifferentIndexes = vector<int> ();
    for(auto it = nonCommonIndexes.cbegin(); it != 
nonCommonIndexes.cend(); ++it){
        sizeDimensionsDifferentIndexes.push_back(it->second);
    }

    vector<int> sizeDimensionsCommonIndexes = vector<int> ();
    for(auto it = commonIndexes.cbegin(); it != commonIndexes.cend();
++it){
        sizeDimensionsCommonIndexes.push_back(it->second);
```

```
    }

    prod_result = UnknownRankedTensor<T>(sizeDimensionsDifferentIndexes);

    shared_ptr<vector<T>> newData =
make_shared<vector<T>>(prod_result.get_n_total_elements());
    for (int i = 0; i < prod_result.get_n_total_elements(); i++)
        newData->at(i) = 0;
    prod_result.setData(newData);

    vector<vector<int>> vectorFactorsIndexes = vector<vector<int>>
(factors.size());
    for(int i = 0; i < (int) vectorFactorsIndexes.size(); i++){
        vectorFactorsIndexes[i] =
vector<int>(factors[i].getTensor().getSizeDimensions().size());
    }

    vector<int> resultIndexes =
vector<int>(prod_result.getSizeDimensions().size());

    map<int, int> totalIndexes = mapOfDifferentIndexes;
    totalIndexes.insert(mapOfEqualIndexes.begin(),
mapOfEqualIndexes.end());
    vector<int> spaceTotalIndexes, sizeTotalIndexes;
    for(map<int,int>::iterator it = totalIndexes.begin(); it !=
totalIndexes.end(); ++it) {
        spaceTotalIndexes.push_back(it->first);
        sizeTotalIndexes.push_back(it->second);
    }

    vector<int> spaceDifferentIndexes;
    for(map<int,int>::iterator it = mapOfDifferentIndexes.begin(); it !=
mapOfDifferentIndexes.end(); ++it) {
            spaceDifferentIndexes.push_back(it->first);
    }

    recursiveProduct(sizeTotalIndexes, spaceTotalIndexes, 0, prod_result,
vectorFactorsIndexes, resultIndexes, spaceDifferentIndexes);
    prod_result.printTensor();

    return prod_result(vectorDifferentIndexes);
}
```

We can see that this method uses all the attributes obtained by the operator * and it applies the product between all the saved factors by using the common and the uncommon indexes. Eventually it produces and returns the final *TensorWithIndexes* results.

Initially the tensor of type *UnknownRankedTensor* has been initialized with a vector of sizeDimensions within the map of uncommon indices and all elements will be equal to *zero*.
This method has to iterate into nested loops the whole indexes of the factors: the common indexes and the uncommon indexes saved into the attributes.
We cannot know at compile time the number of indexes to iterate in a series of nested for loops, so a **recursive** auxiliary function called *recursiveProduct* was needed.
The ***recursive private method recursiveProduct*** is implemented in the following way:

- It takes in input some arguments prepared from the first instructions of *applyProduct*:
  - the vector of integer (called *sizeTotalIndexes*) which contains the dimensions of each index
  - the vector of integers (called *spaceTotalIndexes*) which contains the identifier spaces of each index
  - the size_t index (called *index*) which will be shifted across the vectors to iterate recursively
  - the UnknownRankedTensor *resultInput* (which will be basically the attribute *prod_result*) produced by the product to save
  - a vector of vectors of integers (called *vectorFactorsIndexes*) which will be used for take care of the positions of the various indexes on each tensor in order to retrieve in the correct order the elements. Initially it's empty, but during the recursive function it will be filled with the indexes in the correct positions. In this way, the computation of Einstein's Formalism will retrieve the element in the correct order of indexes for each of the factors.
  - a vector of integers (called *resultIndexes*) which will be used to store in the correct order the elements to the tensor result. Like the previous argument, it is initially empty, but on the innermost loop it will be filled with the iterated indexes in the correct position in order to save the results without errors
  - a vector of integer (called *spaceDifferentIndexes*) which contains the order of the indexes of the resulting tensor
- On the *base case* of the recursion on the vector sizeTotalIndexes the functions returns
- *Otherwise* it applies a for loop in which it saves the index iterated in the correct position for applying correctly the product and it calls a recursive call in order to obtain a series of nested loops of a number equal to the number of indexes
- In the case of the *innermost loop* will be applied the Einstein Notation formula with the usage of the methods *set* and *get* of the *UnknownRankedTensor* class and the usage of the *auxiliary private method* called ***prod***

```
template <typename T>
void MultiplierTensor<T>::recursiveProduct(vector<int> sizeTotalIndexes,
vector<int> spaceTotalIndexes, size_t index, UnknownRankedTensor<T>
```

```
resultInput, vector<vector<int>> vectorFactorsIndexes, vector<int>
resultIndexes, vector<int> spaceDifferentIndexes){
    if (index >= sizeTotalIndexes.size())
        return;

    for (int i = 0; i < sizeTotalIndexes[index]; ++i) {
        for(int j = 0; j < (int)factors.size(); j++){
            vector<Index> indexes = factors[j].getVectorIndexObject();
            for(int z = 0; z < (int)indexes.size(); z++){
                if(spaceTotalIndexes[index] == indexes[z].getSpace())
                    vectorFactorsIndexes[j][z] = i;
            }
       }

       for(int j = 0; j < (int)spaceDifferentIndexes.size(); j++){
            if(spaceDifferentIndexes[j] == spaceTotalIndexes[index])
                resultIndexes[j] = i;
        }

        recursiveProduct(sizeTotalIndexes, spaceTotalIndexes, index + 1,
resultInput, vectorFactorsIndexes, resultIndexes, spaceDifferentIndexes);

        if (index == sizeTotalIndexes.size() - 1) {
            resultInput.set(resultInput(resultIndexes) +
prod(vectorFactorsIndexes), resultIndexes);
        }
    }
}
```

The ***auxiliary private method prod*** used is very simple because it iterates the vector of factors and it computes the product part of Einstein's Formalism.
It exploits the vector of vectors of integers variable obtained from the *recursiveProduct* function in order to retrieve the correct elements every time the product is requested.

```
template <typename T>
T MultiplierTensor<T>::prod(vector<vector<int>> vectorFactorsIndexes) {
    T elem = 1;
    for(int i = 0; i < (int) factors.size(); i++){
        elem *= factors[i].getTensor()(vectorFactorsIndexes[i]);
    }
    return elem;
}
```

Introduction of the tensors of rank zero

Initially the concept of **tensor of rank zero** was set aside as not considered important and, if a zero-rank tensor was created, then an exception was thrown.
Subsequently, with the introduction of the product of tensors and Einstein's formalism within this project, the case study of a zero-rank tensor was managed because we wanted to represent a **trace** as result, a *tensor* which only contains *one element* and *no indexes*, as result of a product in which the Einstein's Formalism deletes all the indexes

A ***tensor of rank zero*** can be created by using the constructor without inserting a list or by inserting an empty vector of integer:

```
vector<int> emptyVector = {};
UnknownRankedTensor<int>zeroRankTensorU1 =
UnknownRankedTensor<int>(emptyVector);
RankedTensor<int, 0> zeroRankTensorR1 = RankedTensor<int, 0>(emptyVector);

UnknownRankedTensor<int> zeroRankTensorU2 = UnknownRankedTensor<int>();
RankedTensor<int, 0> zeroRankTensorR2 = RankedTensor<int, 0>();
```

These tensors will keep their type, they will save the rank zero and inside the vectors of *sizeDimensions* and *strides* the value one will be inserted because this structure will contain only one element of type T:

```
Tensor's rank: 0
Tensor's dimensions: 1
Tensor's strides: 1
Total number of element of the tensor: 1
Tensor's data: 326
```

We also **handled** the **case for rank zeros** for the various methods offered by the tensor in this way:

- *insertRandomData*: it works like in the other tensors and it fill the space for the only element with a random element of an arithmetic type T
- *getIterator*: a tensor of rank zero can't have an iterator because it has rank zero and only one element to iterate
- *printData* and *printTensor*: it doesn't use an iterator and it prints the only element
- *getters* and *setters*: these methods don't need updates for the this introduction
- *get*: when it is applied to a tensor of rank zero, this method will retrieve the only element by having as input an empty vector of integers, no input parameters or an integer vector which contains only a zero
- *set*: when it is applied to a tensor of rank zero, this method will set the only element by having as input the element to be setted and an empty vector of integers or an integer vector which contains only a zero
- *fix* and *fix_copy*: this tensor can't use these methods, so they throw an exception
- *window* and *window_copy*: it can't use these methods, so they throw an exception

- *flattening* and *flattening_copy*: these method works normally and they will build a tensor with rank one which contains the only element of the tensor of rank zero
- *get* (for creating *tensorWithIndexes*): it creates a tensorWithIndexes without index objects into the vector attribute

The **sum** can be applied only between tensors with the same rank, so a **sum between tensors of rank zero** (with the association of their *TensorWithIndexes* objects) can be computed. This case was managed by avoiding the use of iterators for the sums of tensors with rank zero because they can't use them.
This is managed as a special case for the operator + and the result will be another tensorWithIndexes with an associated tensor of rank zero which contains the sum between the element of the first addend and the element of the other addend.

The **product** with **tensors of rank zero** as a factor will throw **exceptions** because this is a particular case that is not requested from the delivery of the exercise. For this reason we have decided to launch exceptions whenever in the two operators * compare a tensor of rank zero as an argument.

Tests

We have made a lot of **tests** for see whether the implemented classes and methods work correctly. Eventually we have left a **main file** which contains a test of each of the functionality inserted in this project.

In this main we have initially created three tensors:

- Two tensors of *UnknownRankedTensor* type with type *int* and *float*
- One tensor of *RankedTensor* type with type *int*

For all the three tensors we apply the ***insertRandomData*** method, we use the ***printTensor*** method and then we show their *sizeDimensions* and *strides* vectors by using the corresponding ***getters***.
Then we have decided to work only on the two tensor of integers of different types in order to test all the other methods for the two tensor classes.
We have applied to these tensors all the methods: ***get***, ***set***, **window**, ***window_copy***, ***fix***, ***fix_copy***, ***flattening*** and ***flattening_copy***. For each method we have saved and printed the results. Subsequently we have tried the two ***getIterators*** method for the two tensors and then we have tested the iterators method *hasNext* and *next* for printing the whole data's tensor or for printing the data along only one index by keeping the others fixed.

Then we have applied the ***sum*** between *three tensorWithIndexes objects* with the ***operator +***.
We have created three new tensors (two *UnknownRankedTensor* and one *RankedTensor*) with the same dimensions and then we have created three corresponding *tensorWithIndexes* objects with the same list of indexes {i, j, k} with the same associated integer spaces (0, 1, 2).
We have saved the tensorWithIndexes result and we have printed the associated tensor.

Then we have tried to applied the **product** between *three tensorWithIndexes* objects, so we have created three new tensors (two *UnknownRankedTensor* and one *RankedTensor*) with different dimensions and we have created three different *tensorWithIndexes* objects with different input indexes. We have to be careful to associate each index to a position in which the dimension is the same for each object, otherwise it prints an exception.
With a product between three tensor we can see how they are working the two ***operators** **** and the method ***applyProduct*** from the multiplierTensorResult.
After the *applyProduct* invoking we have saved the *tensorWithIndexesResult* and we have printed the **final result** of the product through the methods ***getTensor*** and ***printData***.
We have also created another *tensorWithIndexes* object in order to apply another **product** between three tensors, but in this case we want to check whether the result is a **trace**.

Eventually, we have tested an example of *flattening* and *flattening_copy* of some tensor with rank equal to zero. Then we have applied a **sum** with **tensor with rank zero** as **addends**.

References

[1] *Tensor*. (n.d.). Wikipedia. Retrieved April 17, 2023, from https://en.wikipedia.org/wiki/Tensor

[2] *What is a Tensor? Simple Definition, Ranks*. (n.d.). Statistics How To. Retrieved April 17, 2023, from https://www.statisticshowto.com/what-is-a-tensor/

[3] *What do you mean by Tensor and Explain about Tensor Datatype and Ranks?* (n.d.). i2tutorials. Retrieved April 17, 2023, from https://www.i2tutorials.com/what-do-you-mean-by-tensor-and-explain-about-tensor-datatype-and-ranks/

[4] *Parameterized Class :: Chapter 6. Class Diagrams: Advanced Concepts :: UML :: Programming*. (n.d.). eTutorials.org. Retrieved April 18, 2023, from http://etutorials.org/Programming/UML/Chapter+6.+Class+Diagrams+Advanced+Concepts/Parameterized+Class/

[5] *Namespaces (C++)*. (2021, August 2). Microsoft Learn. Retrieved April 28, 2023, from https://learn.microsoft.com/en-us/cpp/cpp/namespaces-cpp?view=msvc-170

[6] *Inheritance in C++*. (2023, February 17). GeeksforGeeks. Retrieved April 28, 2023, from https://www.geeksforgeeks.org/inheritance-in-c/

[7] *std::shared_ptr - cppreference.com*. (2023, March 2). C++ Reference. Retrieved April 28, 2023, from https://en.cppreference.com/w/cpp/memory/shared_ptr

[8] *How to: Create and use shared_ptr instances*. (2021, August 2). Microsoft Learn. Retrieved April 28, 2023, from https://learn.microsoft.com/en-us/cpp/cpp/how-to-create-and-use-shared-ptr-instances?view=msvc-170

[9] *The Rule of Five in C++ | Explained*. (n.d.). CodersLegacy. Retrieved April 30, 2023, from https://coderslegacy.com/c/rule-of-five-cpp/

[10] *The rule of three/five/zero*. (2023, February 4). cppreference.com. Retrieved April 30, 2023, from https://en.cppreference.com/w/cpp/language/rule_of_three

[11] Virtual Destructor. (2023, February 20). GeeksforGeeks. Retrieved April 30, 2023, from https://www.geeksforgeeks.org/virtual-destructor/

[12] Iterator pattern. (n.d.). Wikipedia. Retrieved May 1, 2023, from https://en.wikipedia.org/wiki/Iterator_pattern

[13] Introduction to Iterators in C++. (2022, April 28). GeeksforGeeks. Retrieved May 1, 2023, from https://www.geeksforgeeks.org/introduction-iterators-c/

[14] Iterators in Java. (2023, April 22). GeeksforGeeks. Retrieved May 1, 2023, from https://www.geeksforgeeks.org/iterators-in-java/

[15] *The C++ Standard Template Library (STL)*. (2023, March 19). GeeksforGeeks. Retrieved May 1, 2023, from https://www.geeksforgeeks.org/the-c-standard-template-library-stl/

[16] Einstein notation. (n.d.). Wikipedia. Retrieved May 8, 2023, from https://en.wikipedia.org/wiki/Einstein_notation

[17] *Tensor Diagram Notation–Tensor Network*. (n.d.). Tensor Network. Retrieved May 8, 2023, from https://tensornetwork.org/diagrams/

[18] Hirsch, K. (n.d.). Trace (linear algebra). Wikipedia. Retrieved May 8, 2023, from https://en.wikipedia.org/wiki/Trace_(linear_algebra)