Lab 8 – Traits and policies

Exercise 1. Traits numeric_limits

Implement template function **info**, that for given argument prints information about its type:

- if it is signed or unsigned
- if it is an integer type
- minimal and maximal value

Example

```
info(1);
info(2.0f);
info(3U);
```

Expected output:

```
signed,  integer, min : -2147483648 max : 2147483647
signed, not integer, min : 1.17549e-38 max : 3.40282e+38
unsigned,  integer, min : 0 max : 4294967295
```

Exercise 2. Type traits for Vector

For given class Vector<T,N> define template class vector_traits<T> that for given type T defines:

- How arguments of type T are passed from or to methods get and set of a Vector<T>.
  By default they should be passed by const reference, but for types **int** and **double** by value.
- The type of scalar in
  Vector operator * (scalar, const Vector & x)
  By default it should be T, but for std::string it should be int.
- Operation used in operator *. By default it should be multiplication by scalar, but strings should be multiplied like this 3 * [„to”, „A”] → [„tototo”,”AAA”],
- Static method that returns the default value of type T :
  zero for arithmetic types and "0" for strings.

Modify class template Vector<T,N> but do not define its specializations.

Exercise 3. Policies

To class template Vector<T,N> add template parameter P that will pass policies:

- **init policy:** decides if default constructor initializes vector with default values (zeroes).
- **check policy:** defines if methods get and set are checking indices and how they react on the out of bound error.

Create policies: **safePolicy** (it initializes, checks indices and throws exception) and **fastPolicy** (do not initializes and do not checks indices).

Is there a way to easily define all combinations of policies?

```
Vector<int, 5, SafePolicy> a;
a.set(6, 9); // throws an exception
Vector<double, 4, FastPolicy> b;
b.set(6, 0);   // the result is unspecified
```

Exercise 4. Expressions unfolding

For vectors a, b, c and d when computing

z = a+b+c+d;

it is not optimal to evaluate immediately each sum and create temporary objects.
It is better to define lazy evaluation of sums so that the expression will be evaluated internally almost as

```
for(int i =0; i<N; i++)
    z[i] = a[i]+b[i]+c[i]+d[i];
```

This can be done by defining auxiliary types that will hold references to left and right hand sides of operator and perform computations only if needed. They should provide operator [] and conversion to vector.

Expression a+b+c+d should be converted to something like

```
AddNode(AddNode(AddNode(a,b), c), d)
```

Implement the same behaviour for the subtraction of two vectors and the multiplication of vector by constant.
As a starting point you can use file **ex_8_4_Vector_operators.cpp**.