Vector – Implementation description

For the test in Moped your **Vector** class should be defined in only one header file (**vector.h**), i.e., a separation into a .h and .cpp file, as otherwise usual with classes, is not required here. The use of only one header file facilitates the transition to a template version of the **Vector** class.

For testing you may only bring the basic functionality mentioned below consisting of basic functionality, iterators and templates. In particular, no solutions of old test specifications, sample tests or the like may be included. So, if necessary, save an intermediate state before you implement further methods for practice purposes.

Note the deadline for uploading your vector implementation 25.11.2022

To test your implementation you should write your own test routines (e.g. `main` files). Test files are provided every week as a supplement. Note that these are only intended to help you and **do not guarantee** that your vector will work correctly.

1 Basic Functionality

1.1 Instance variables

The `Vector` class has the following **instance variables**..

size_t sz: Contains the number of elements in the **Vector**.

size_t max_sz: Contains the maximum number of elements that are possible (capacity of the **Vector**).

double* values: Points to a array containing the elements of the **Vector**.

Hints:

You may introduce an additional variable **static constexpr size_t min_sz**, which defines a minimum size (e.g. 5) for max_sz. Whether you allow empty vectors or define a minimum size is largely a matter of personal taste in our context. Both approaches have their advantages and disadvantages and there are specific special cases in the code that need to be handled. These special cases just occur in different places. You can define a typalias **using value_type = double;** and then use **value_type** instead of **double** everywhere it fits. This will make it easier for you to transition to templates later.

1.2 Constructors/Destructor

The `Vector` class has the following **constructors and one destructor**.

Default constructor: Liefert einen leeren **Vector**.

Copy Constructor: Liefert einen **Vector** mit demselben Inhalt.

Constructors with the following parameter lists

(size_t n): Returns a **Vector** with space for n elements..

(std::initializer_list<double>): Returns a **Vector** with specified content.

Destruktor: Frees allocated memory.

Avoid memory leaks and be aware of special cases like Vector(0) and Vector({}). You can identify such problems in your code, for example, using `valgrind`, which is easy to install on the virtual machine.

1.3 Member Functions

The `Vector` class has the following **member functions**.

Copy assignment operator: The `this` object takes the values from the parameter. (Necessary because of the use of dynamically allocated memory).

size_t size() const: Returns number of saved elements.

bool empty() const: Returns `true` if the **Vector** is empty, otherwise `false`.

void clear(): Deletes all elements from **Vector**.

void reserve(size_t n): Capacity of the **Vector** is increased to n if it is not already at least this large.

void shrink_to_fit(): Capacity is reduced to number of elements.

void push_back(double x): Adds a copy of `x` to the end of the **Vector**.

void pop_back(): Removes the last element in the **Vector**. Throws an `std::runtime_error` exception if the **Vector** was empty.

double& operator[](size_t index): Returns the element at the given position (index). If index is out of bounds, throws an `std::runtime_error` exception

const double& operator[](size_t index) const: Returns the element at the given position (index). If index is out of bounds, throws an `std::runtime_error` exception

size_t capacity() const: Returns current capacity of the **Vector**.

1.4 Output format

The `vector` class has the following **output format**, which is mandatory.

ostream& operator<<(ostream&, const Vector&): Outputs: [Element1, Element2, Element3].

Example Vector x({1,2,3,4}) → [1, 2, 3, 4]

Use of **friend** for implementation of **operator**<< is allowed.

2 Iterators

In order to use iterators of STL algorithms, some type aliases must be created for the iterators. The easiest way is to define them at the beginning of the `Vector` class. Make sure that your `Vector` class only uses the data types from the `using` declarations.

```
class Vector{
public:
  class ConstIterator;
  class Iterator;
  using value_type = double;
  using size_type = std::size_t;
  using difference_type = std::ptrdiff_t;
  using reference = value_type&;
  using const_reference = const value_type&;
  using pointer = value_type*;
  using const_pointer = const value_type*;
  using iterator = Vector::Iterator;
  using const_iterator = Vector::ConstIterator;
private:
  //Instance variables
public:
  //Member Functions
class Iterator {
    public:
      using value_type = Vector::value_type;
      using reference = Vector::reference;
      using pointer = Vector::pointer;
      using difference_type = Vector::difference_type;
      using iterator_category = std::forward_iterator_tag;
    private:
    //Instance variables
    public:
    //Member Functions
  };
  class ConstIterator {
    public:
      using value_type = Vector::value_type;
      using reference = Vector::const_reference;
      using pointer = Vector::const_pointer;
      using difference_type = Vector::difference_type;
      using iterator_category = std::forward_iterator_tag;
    private:
    //Instance variables
    public:
    //Member Functions
  };
};
```

2.1 Extending the Vector

Extend your **Vector** class with **begin()** and **end()** member functions.

iterator begin(): Returns an iterator to the first element in the **Vector**. If the **Vector** is empty, the returned iterator corresponds to the end iterator.

iterator end(): Returns an iterator to the virtual element after the last element in the **Vector**.

const_iterator begin() const: Returns an iterator to the first element in the **Vector**. If the **Vector** is empty, the returned iterator corresponds to the end iterator.

const_iterator end() const: Returns an iterator to the virtual element after the last element in the **Vector**.

2.2 Iterator

The class **Iterator** has the following **instance variables**.

pointer ptr: Points to an element in **Vector**.

The class **Iterator** has the following **constructors**.

Default: Returns an iterator on **nullptr**.

Parameter list (pointer ptr): Returns an iterator which sets the instance variable to **ptr**.

The class **Iterator** has the following **member functions**. Which methods should be **const**?

reference operator*() const?: Returns the value of the value referenced by **ptr**.

pointer operator->() const?: Returns a pointer to the referenced value.

bool operator==(const const_iterator&) const?: Compares the pointers for equality. (A global function may be a better choice).

bool operator!=(const const_iterator&) const?: Compares the pointers for inequality. (A global function may be a better choice).

iterator& operator++() const?: (Prefix) Iterator points to next element and (a reference to it) is returned.

iterator operator++(int) const?: (Postfix) Iterator points to next element. Copy of iterator before increment is returned.

operator const_iterator() const?: (Type conversion) Allows to convert **Iterator** to **ConstIterator**.

2.3 ConstIterator

The **ConstIterator** class has the following **instance variables**.

pointer ptr: Points to an element in **Vector**.

The **ConstIterator** class has the following **constructors**.

Default: Returns a ConstIterator on **nullptr**.

Parameter list (pointer ptr): Returns a ConstIterator which sets the instance variable to **ptr**.

The class **ConstIterator** has the following **member functions**. Which methods should be **const**?

reference operator*() const?: Returns the value of the value referenced by **ptr**.

pointer operator->() const?: Returns a pointer to the referenced value.

bool operator==(const const_iterator&) const?: Compares the pointers for equality. (A global function may be a better choice).

bool operator!=(const const_iterator&) const?: Compares the pointers for inequality. (A global function may be a better choice).

const_iterator& operator++() const?: (Prefix) Iterator points to next element and (a reference to it) is returned.

const_iterator operator++(int) const?: (Postfix) Iterator points to next element. Copy of iterator before increment is returned.

2.4 Member functions `insert` and `erase`

The member functions **insert** and **erase** can be copied from here.

```
iterator insert(const_iterator pos, const_reference val) {
  auto diff = pos-begin();
  if (diff<0 || static_cast<size_type>(diff)>sz)
    throw std::runtime_error("Iterator out of bounds");
  size_type current{static_cast<size_type>(diff)};
  if (sz>=max_sz)
    reserve(max_sz*2); //Attention special case, if no minimum size is defined
  for (auto i {sz}; i-->current;)
    values[i+1]=values[i];
  values[current]=val;
  ++sz;
  return iterator{values+current};
}

iterator erase(const_iterator pos) {
  auto diff = pos-begin();
  if (diff<0 || static_cast<size_type>(diff)>=sz)
    throw std::runtime_error("Iterator out of bounds");
  size_type current{static_cast<size_type>(diff)};
  for (auto i{current}; i<sz-1; ++i)
    values[i]=values[i+1];
  --sz;
  return iterator{values+current};
}
```

In order for the **insert** and **erase** methods to work, the following must be implemented as well

```
friend Vector::difference_type operator-(const Vector::ConstIterator& lop,
                                         const Vector::ConstIterator& rop) {
  return lop.ptr-rop.ptr;
}
```

3 Templates

To make your **vector** class a template, it is recommended to proceed as follows:

1. Class **Vector** becomes a template with a type parameter

   ```
   template<typename T>
   class Vector {...};
   ```

2. Replacing the `double` datatype as element type with `T` (less work if you have already neatly used the type aliases, otherwise now a good way to catch up).

3. The definitions of the template methods also go into the header file (vector.h). It is easiest if the methods are defined right inline (within the class definition). These definitions are needed by the compiler for the instantiation.

4. Fix any errors and test with different data types.