Advanced Programming in C++

Exercise – fixed size vector

This is an exercise on design of a fairly simple sequence container, named fixed_size_vector.

A fixed_size_vector object shall have a fixed size and basically behave as a simple array type, i.e. be a behaviourless aggregate. A rudimentary definition, with some member and a non-member declarations, is given below.

```
template<typename T, unsigned int N>
struct fixed_size_vector
{
  // Types

  T elements_[N];

  // Construct/copy/destroy

  void fill(const T& t);

  void swap(fixed_size_vector<T, N>& other

  // Iterators

  // Capacity

  // Element access

};

template<typename T, unsigned int N>
void swap(fixed_size_vector<T, N>& x, fixed_size_vector<T, N>& y);

// Comparisons (==, !=, <, >, <=, >=)
```

Storage

The elements stored by a fixed_size_vector are to be kept in the public member elements_, an array of size N storing elements of type T.

The implementation shall support fixed_size_vector objects with size 0. The dimension of an array is not allowed to be 0, so in that special case the dimension cannot be set to N. You have to do some fix in the definition of elements_ for handling this.

Types

Some library component, e.g., require other components to supply specific type definitions. The following types are to be declared as nested types of fixed_size_vector:

value_type	same as T
reference	reference to T (i.e., reference to value_type)
const_reference	constant reference to T
pointer	pointer to T
const_pointer	constant pointer to T

size_type	the type of the template parameter N
difference_type	declare as ptrdiff_t
iterator	Your choice – fixed_size_vector can support random access iterators
const_iterator	constant iterator
reverse_iterator	use std::reverse_iterator
const_reverse_iterator	use std::const_reverse_iterator

Construct, copy and destroy

The requirements on fixed_size_vector regarding construction, copying and destruction are:

- if type T have a default constructor, it shall be invoked for each element in elements_, otherwise no explicit initialization is to be performed
- copy construction shall be a member by member copy
- copy assignment shall be a member by member assignment
- if type T have a default destructor it shall be invoked for each element in elements_, otherwise no explicit destruction is to be performed

Given operations

void fill(**const** T& t);
    Assign each element in elements_ to t.

void swap(fixed_size_vector<T, N>& other);
    swap the elements in **this** and other pair-wise.

void swap(fixed_size_vector<T, N>& x, fixed_size_vector<T, N>& y);
    swap corresponding elements in x and y.

Hint: There are standard algorithms suitable for implementing these operations.

Iterators

fixed_size_vector shall have the following member functions returning iterators:

begin()	shall return an iterator pointing at the first element
end()	shall return an iterator pointing at the position past the last element
rbegin()	shall return a reverse iterator pointing at the position past the last element
rend()	shall return a reverse iterator pointing at the first element

If N == 0, begin() == end() == unique value (your choice).

Capacity

The following three operations related to capacity shall be defined:

size()	invariant, N
max_size()	N
empty()	**true** if N == 0, otherwise **false**

N.b., there is no relation to whether any values are stored.

Element access

The following operations for accessing elements are to be defined:

operator[*pos*]	unchecked access to element access at given index (*pos*)
at(pos)	checked access to element at pos, throws std::out_of_range if pos is not within range [0, N-1).
front()	returns reference to element at position 0
back()	returns reference to element at position N-1
data()	returns a pointer to elements_[0] (the address to elements_[0])

For a zero-sized fixed_size_vector, the return value of data(), and the effect of calling front() and back(), can defined as you please.

Comparisons

The following operations for comparing fixed_size_vector objects shall be defined:

operator==(**const** fixed_size_vector<T, N>& lhs, **const** fixed_size_vector<T, N>& rhs)
compare corresponding elements in lhs and rhs, and return **true** if all element pairs are equivalent, **false** otherwise

operator!=(**const** fixed_size_vector<T, N>& lhs, **const** fixed_size_vector<T, N>& rhs)
compare corresponding elements in lhs and rhs, and return **true** if at least one element pair is not equivalent, **false** otherwise. Implement by using **operator**==.

operator<(**const** fixed_size_vector<T, N>& lhs, **const** fixed_size_vector<T, N>& rhs)
compare the elements in lhs and rhs lexicographically, and return **true** if lhs is lexicographically less than rhs.

operator>(**const** fixed_size_vector<T, N>& lhs, **const** fixed_size_vector<T, N>& rhs)
compare the elements in lhs and rhs lexicographically, and return **true** if lhs is lexicographically greater than rhs. Implement by using **operator**<.

operator<=(**const** fixed_size_vector<T, N>& lhs, **const** fixed_size_vector<T, N>& rhs)
compare the elements in lhs and rhs lexicographically, and return **true** if lhs is lexicographically less or equal than rhs. Implement by using **operator**<.

operator>=(**const** fixed_size_vector<T, N>& lhs, **const** fixed_size_vector<T, N>& rhs)
compare the elements in lhs and rhs lexicographically, and return **true** if lhs is lexicographically greater or equal than rhs. Implement by using **operator**<.

Hint: There are standard algorithms suitable for implementing **operator**== and **operator**<.