NPRG041 – 2024/25 Winter – Labs MS – **Small Assignment C11**

Flexible Array II

Within this assignment, we finalize our representation of the flexible array container. We first preserve the existing code and extend it by implementing advanced constructors and assignment operators. Next, we program our own custom iterators for this container, in particular, at the level of the highest category of random access iterators.

Regarding the aforementioned advanced constructors and assignment operators, we implement the following standard four functions:

- `Array(const Array<Element>& other)`:
  copy constructor
- `Array(Array<Element>&& other) noexcept`:
  move (stealing) constructor
- `Array<Element>& operator=(const Array<Element>& other)`:
  copy assignment operator
- `Array<Element>& operator=(Array<Element>&& other) noexcept`:
  move (stealing) assignment

Let us emphasize that (not only) the copy assignment operator is again expected to ensure atomic behavior, i.e., that in case of failure, the original content (of the flexible array being assigned to) will fully be preserved. Additionally, both the assignment operators should include a preventive protection to handle situations where we might attempt to assign an instance to itself. Following the move constructor and move assignment operator, we will also implement a global function `void swap(Array<Element>& array_1, Array<Element>& array_2) noexcept`. It will allow (not only us) to easily swap the contents of two instances of flexible arrays.

Our custom iterator for the flexible array container is expected to be implemented in the form of a public inner class template `iterator_base<bool Constant>` within our array class. Its only parameter `Constant` will be a flag `false` or `true` suggesting whether the iterator should work over a modifiable or constant flexible array. More precisely, whether it should return references to modifiable or constant elements. From a practical point of view, we need to offer both these variants of behavior, and the approach with template code will help us avoid unnecessary duplication of code.

In order not to unnecessarily burden the users of our iterators with internal details, however, we will offer two public type aliases in the form of `iterator` for `iterator_base<false>` and `const_iterator` for `iterator_base<true>`, respectively. The flexible array class will then offer the following standard methods for creating iterator instances pointing to the first element and beyond the last one, respectively:

- `iterator begin()`, `const_iterator begin() const`, and `const_iterator cbegin() const`:
  returns an iterator instance in the modifiable or constant variant that will point to the first element of our flexible array (if there is any, otherwise beyond the end)
- `iterator end()`, `const_iterator end() const`, and `const_iterator cend() const`:
  analogously, returns an iterator instance pointing beyond the current end of our array

Let us now look at the implementation of the base iterator class itself. In order to be able to use both the variants of our iterators in connection with standard algorithms, it is first necessary to describe their behavior and capabilities using the following public type aliases. Tags for the individual categories of iterators can be found in the `<iterator>` library:

- `using iterator_category = std::random_access_iterator_tag`
- `using value_type = Element`
- `using pointer = Element*`
- `using reference = Element&`
- `using difference_type = std::ptrdiff_t`

Items `value_type`, `pointer`, and `reference` will only work for the modifiable variant, though. We thus replace them using the construct `std::conditional_t<bool B, class T, class F>`, which simply selects type `T` or `F` according to the actual `true` / `false` value of parameter `B`. The construct itself can be found in the `<type_traits>` library. We can then use the same trick to choose the correct type to preserve a reference or pointer to the flexible array itself within the data members we will need to remember.

First, we focus on implementing the required methods at the forward iterator level:

- `bool operator==(const iterator_base& other) const`: tests the equality of our iterator with respect to another one over the same flexible array, i.e., detects whether they both point to the same logical position
- `bool operator!=(const iterator_base& other) const`: tests the inequality of both iterators, i.e., that they point to different positions
- `iterator_base& operator++()`: pre-increments our iterator, i.e., moves the current logical position forward by 1
- `iterator_base operator++(int)`: analogously for post-increment
- `reference operator*() const`: dereferences our iterator, i.e., returns a reference to the element the iterator is currently pointing to
- `pointer operator->() const`: returns a pointer to the element the iterator is currently pointing to

On top of the operators already mentioned, we add the following two to reach the level of bidirectional iterators:

- `iterator_base& operator--()`: performs iterator pre-decrement
- `iterator_base operator--(int)`: analogously for post-decrement

Next, we extend the offered functionality up to the level of random access iterators:

- `iterator_base operator+(difference_type n) const`: returns a new iterator instance that will point to a position shifted by the appropriate number of elements, forward for a positive number and backward for a negative number
- Analogously for `operator-`
- `difference_type operator-(const iterator_base& other) const`: calculates the distance of two iterators, i.e., returns the number of elements between the other iterator and ours
- `iterator_base& operator+=(difference_type n)`: shifts our iterator by the appropriate number of positions and returns a reference to it
- Analogously for `operator-=`
- `reference operator[](difference_type n) const`: returns a reference to an element of the flexible array that occurs a given number of positions ahead relative to the current position our iterator is currently pointing to
- `bool operator<(const iterator_base& other) const`: mutually compares our and another passed iterator, i.e., detects whether our iterator points to a lower position than the second one
- Analogously for `operator<=`, `operator>`, and `operator>=`

We have implemented all the above operators as member functions of our iterator class. We also need to support expressions in the form `n + it`, though, in order to allow shifting a given iterator `it` forward by the appropriate number of positions `n`, but in a variant with the order of the operands swapped. This can only be achieved through a global function `iterator_base operator+(difference_type n, const iterator_base& it)`.

For practical reasons, it is also advisable to be able to convert an iterator in the modifiable variant to the constant one, i.e., retype `iterator` to `const_iterator`. Of course, in this direction only. We achieve the required behavior by adding a member function for the conversion operator in the form `operator iterator_base<true>() const`. The only task is to return a newly created instance of the appropriate iterator.

Let us add that the responsibility for the correct use of iterators is fully transferred to the users. This means that the following situations must be avoided at all times: using iterators across different instances

of flexible arrays, using invalidated iterators (due to operations that modified a given flexible array as such), accessing invalid positions (e.g., beyond the flexible array end), dereferencing invalid positions, etc. In such and similar cases, the behavior of our iterators will be undefined, and we will not detect or handle the relevant situations in any way.

Finally, we move the entire implementation of our flexible array and iterators into our custom namespace called `lib`.

Again, submit only the `Array.h` header file and follow the usual assignment requirements. The goal of this task is to demonstrate the ability to design and work with copy and move constructors and assignment operators, implement custom iterators, work with nested templates, conversion operators, and also custom namespaces.