NPRG041 – 2025/26 Winter – Labs MS – **Small Assignment C11**

Flexible Array II

Within this assignment, we finalize our representation of the flexible array container. We entirely preserve the existing code and gradually extend it with the implementation of advanced container constructors, advanced methods for inserting new elements, as well as additional helper methods for the manipulation with elements. Last but not least, we also program our own custom iterators for this array, in particular at the level of the random access iterators category.

At first, however, we adjust the implementation of our array in a way to make its use user-friendly and also compatible with standard libraries. This will significantly expand the potential of its use. In other words, it will no longer be just some proprietary solution of ours, but a fully-fledged container with which even standard algorithms or ranges can work. We will actually focus on those right in the next, i.e., the last small assignment.

Let us start with placing the whole flexible array implementation into a custom namespace called `lib`. Next, we add a couple of public type aliases into the array class itself, through which others will be able to retrieve necessary information about type traits of the managed elements:

- `using value_type = T;`
- `using size_type = std::size_t;`
- `using difference_type = std::ptrdiff_t;`
- `using reference = T&;`
- `using const_reference = const T&;`
- `using pointer = T*;`
- `using const_pointer = const T*;`

In accordance with the previous task, template parameter `T` denotes the type of our elements. Later on, we will also add two additional such aliases, these will relate to our custom iterators.

Another new feature will be a systematic use of concepts and related mechanisms in order to gain better control over particular types `T` that are meaningfully usable for our elements, whether at the level of the array as a whole or its particular selected methods. For this purpose, we define a custom concept `ArrayElement` ensuring at least one form of constructibility `std::copy_constructible`, `std::move_constructible`, or `std::default_initializable`, and, at the same time, destructibility `std::destructible`. All the listed standard concepts can be found in the `<concepts>` library. Thanks to that, the declaration of our flexible array can thus become `template<ArrayElement T> class Array` from now on.

We now focus on three new constructors that will allow for advanced creation of new flexible array instances with non-empty content from the very moment they are created. In all cases, we will consider the default block size to be 10, while the [♦] symbol reminds us of the need to guarantee their atomicity.

- `Array(size_t count, const T& item)` [♦]: creates a new array instance that will contain `count` copies of the provided sample element `item`
- `Array(std::initializer_list<T> items)` [♦]: creates a new array instance into which we insert copies of the elements provided in a given initializer list (library `<initializer_list>`)
- `template<std::input_iterator InputIterator>`
  `Array(InputIterator begin, InputIterator end)` [♦]: creates a new array into which we insert copies of the elements that belong to a range [`begin`, `end`) over some collection (another flexible array, different container, or even a C-style array), determined by a given pair of iterators or even C-style pointers (library `<iterator>`)

The next step is to extend the existing means for inserting new individual elements into our array. We currently have a pair of the `push_back` functions available for this purpose. Since their implementation is practically identical, we will first merge them together, or rather replace them with just a single function template, all because of a relatively straightforward trick based on the universal references and perfect forwarding. Besides that, we will also implement equally useful `emplace_back` function via a variadic template.

- `template<typename Source>`
  `requires std::constructible_from<T, Source&&>`
  `void push_back(Source&& item)` [♦]: via copying or moving (depending on the original category of the passed value in terms of lvalue or rvalue), we insert a given new element at the end of the array; to preserve its original value category within its subsequent forwarding, we will use the `std::forward` function (library `<utility>`)
- `template<typename... Args>`
  `requires std::constructible_from<T, Args&&...>`
  `T& emplace_back(Args&&... args)` [♦]: similarly, we insert a new element at the end of the array, but in this case this element is created by calling a constructor that expects exactly the same parameters as those passed to this function; we again use the same mechanism of perfect forwarding, only with a parameter pack expansion involved, i.e., we will use construct `std::forward<Args>(args)...`; finally, we return a modifiable reference to the inserted element

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 the 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. It is actually not a coincidence that we chose exactly these names, as they correspond to the remaining two aliases, the definition of which we promised in the introduction. 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. We implement them all as inline methods, and, for greater clarity, we also combine their declarations directly with definitions:

- `inline iterator begin()` / `const_iterator begin() const` / `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)
- `inline iterator end()` / `const_iterator end() const` / `const_iterator cend() const`: analogously, returns an iterator instance pointing beyond the current end of the 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 the already mentioned standard algorithms, it is also and not surprisingly necessary to describe their behavior and capabilities using several 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 = T;`
- `using pointer = T*;`
- `using reference = T&;`
- `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 `std::conditional_t<bool B, class T, class F>` construct, 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.

We will design a parameterized iterator constructor as needed. In other words, its exact form is not prescribed. However, with regard to the aforementioned compatibility, it is also necessary to implement a public default constructor. Instances created in this way obviously cannot contain any meaningful information, but that is not important. What matters is that such a parameterless constructor is available.

As for the expected functionality, it will solely be about a variety of operators, moreover, quite trivial in terms of their content. In order to increase the overall clarity and also make our work easier, we will

again connect their declarations directly with definitions. Let us first focus on implementing the required methods at the forward iterator level (tag `std::forward_iterator_tag`):

- `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 (tag `std::bidirectional_iterator_tag`):

- `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 (the resulting tag `std::random_access_iterator_tag` we expected from the beginning):

- `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 a given iterator and the other one
- `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`: performs the `<` mutual comparison of two iterators, 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-mentioned 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, moreover, which can only be defined directly within a friend declaration:

- `friend iterator_base operator+(difference_type n, const iterator_base& it)`

For practical reasons, it is also advisable to be capable of converting 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 this behavior by adding a member function for the conversion operator in the following form:

- `operator iterator_base<true>() const`: returns a newly created instance of a constant iterator that will point to the same position as the source iterator

Let us add that the responsibility for the correct use of iterators is again fully transferred to the users. This means that they must avoid, e.g., the following situations: 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 these situations in any way.

We now return once again to the flexible array itself and add two more functions for manipulating its elements. These will become useful in the last small assignment.

- `void resize(size_t size)`: we change the size of the flexible array to a new size `size`; when decreasing the size, we simply remove the corresponding number of existing elements from the end of the array; when growing, we add new elements to the end; these new elements will, in particular, be created using their default constructor
- `iterator erase(const_iterator first, const_iterator last)`: we remove all the elements from the array that belong to an interval [`first`, `last`) specified by our iterators; we return a modifiable iterator pointing to the first element after the last removed one (if any was removed at all); to reduce the complexity of this operation, we assume that our elements are movable, and, moreover, without exceptions; thanks to this, we do not need to (and we will not) address any potential problems with achieving the atomicity, since they cannot arise under such conditions

At the very end of the assignment, we will also add various static assertions in the form of `static_assert (condition, message)`, i.e., compile-time checks through which we can verify the fulfillment of specifically required capabilities of elements at the level of individual functions. The purpose of these checks is not to further restrict our container as a whole, but merely to identify the assumptions we already implicitly had, and allow the compiler to emit meaningful error messages in case they are not satisfied.

Particularly expected checks, i.e., the affected container functions, checked element capabilities, and error messages are not prescribed and will not be assessed (since it is not possible to do so automatically). You should therefore enrich your code as best as you can in this regard. To do so, you may use the following standard concepts or `requires` conditions in particular:

- `std::copy_constructible`
- `std::constructible_from`
- `std::assignable_from`
- `std::default_initializable`
- `requires(T& t, T&& s) { { t = std::move(s) } noexcept; }`

Again, submit only the `Array.h` header file and follow the usual assignment requirements. The goal of this assignment is to demonstrate the ability to design advanced container constructors and custom iterators, work with nested templates, variadic templates, and perfect forwarding, conversion operators, custom namespaces, as well as concepts and static assertions.