NPRG041 – 2025/26 Winter – Labs MS – **Small Assignment C10**

Flexible Array I

Within this and the next assignment, we propose an implementation of our own generic container, namely a flexible array container. Its implementation will largely be based on the behavior and principles of the standard vector container, but we will deliberately design some of its aspects in a significantly different way. We also take inspiration from the public interface of the standard vector, but we only implement a certain selection of its basic or interesting methods, as implementing all of them would not be beneficial enough for us. In this task, we first focus on the internal storage of such a flexible array, its basic functionality, and copy and move constructors and assignment operators. In the next assignment, we will take care of the container consolidation, advanced container constructors, advanced means of new elements insertion, and, most importantly, we will implement our own iterators.

The goal with our flexible array is to preserve the possibility of gradually inserting new elements, or, on the contrary, removing the existing ones, but always at its end only. In other words, this means that the entire flexible array will always be logically contiguous without any gaps. Compared to the standard vector, however, we will program the internal storage in a way that we will do without the need of having the allocated space in memory for elements themselves that would also have to be contiguous physically. Thanks to this, we will not need time-consuming reallocations, and, moreover, we will be able to guarantee that the location of inserted elements will be immutable throughout their existence in the container. We thus completely eliminate the risk of invalidating pointers, references, or iterators to the individual array elements throughout their entire lifetime, which the standard vector container is not able to guarantee.

The flexible array class, let us simply call it `Array`, will have a single template parameter `T`, being the data type of elements to be inserted. These can be numeric types, as well as any user-defined classes (which we cannot assume much about in advance), or even pointers to them (whether C-style or smart).

We implement the outlined internal storage as a two-level storage, where we first use a standard vector `std::vector` at the first level for storing C-style pointers to blocks of the second level. By a second-level block, we mean an ordinary dynamically allocated C-style array of individual elements. All of these blocks will have the same and fixed size that will remain constant throughout the existence of a given flexible array instance. This will allow us to use a straightforward index arithmetic to access particular elements, where an element at a logical position i will be placed into block (i/s) and its physical position $(i \bmod s)$, where s is the chosen block size (measured as a number of element slots).

A newly created flexible array container will be empty, i.e., it will not contain any elements, and especially no inner blocks. These will then be dynamically added or removed as need be, following the requests of adding, or, on the contrary, removing individual elements. We define the necessary internal mechanisms in such a way that we will always have exactly the number of allocated inner blocks that is absolutely necessary to store the current number of elements. Of course, different or even smarter strategies could be used as well, but we do not want to unnecessarily complicate the whole situation for us.

We now focus on describing details of the expected functionality and the interface of particular member or even global functions. We begin with a basic constructor (additional ones will be added later on) and destructor:

- `Array(size_t block_size)`: creates an empty flexible array instance; the only passed parameter indicates the requested block size; if not specified, we will assume a default size equal to 10 elements
- `~Array() noexcept`: performs flexible array destruction

We then offer the following three member functions to obtain basic information about the current content and capacity of the container:

- `inline size_t size() const`: returns the size of the array, i.e., the number of elements currently inserted into it
- `inline bool empty() const`: returns `true` if only if the array currently contains no elements
- `inline size_t capacity() const`: returns the size of the allocated internal storage space, i.e., the number of elements that all the currently allocated internal blocks are capable of storing

In order to add a new element to the flexible array end or to remove the last element from the end, we offer the following methods. In the case of additions, we will actually offer both copying and moving (stealing) variants. We will also offer a method for removing the entire array content.

- `void push_back(const T& item)` [♦]: adds a new element to the current end of the flexible array; we physically place the element as such to the first unused position in the current (i.e., last) internal block; more precisely, we place a copy of the passed element at this position and become its owner; if there is no more free space in the current block, we first add a new one
- `void push_back(T&& item)` [♦]: the same behavior as the previous variant, we only assume the element is passed by an rvalue reference, and so we just move it to the appropriate position using `std::move`, i.e., without its copying; we become its owner again and the caller may no longer use the original instance
- `void pop_back()`: removes the last element from the current end of the flexible array; since we were its owner in any case, we must take care of its correct manual destruction; if removing the element made the last inner block completely unused, we remove it as well
- `void clear()`: empties the container and so removes all its elements

Even at this point, it should be intuitively clear that adding new elements may not always succeed due to the possibility of dynamic allocation failure in our internal storage. In general, it is entirely up to us how we choose to handle such situations, if at all. Nevertheless, we aim to adhere to the principles applied in the standard containers. This specifically means we ensure the so-called strong exception guarantee. In other words, we ensure atomicity of any potentially unsafe operations. That is, in case of failure, we will restore the internal content of the entire flexible array to the original valid state before the call of the respective function. As for the container usability, any other behavior would in fact only hardly be conceivable. We will therefore adhere to this principle in all our methods where the risk of failure exists. And since there will be more of them, we will always mark them with the already used symbol [♦] for better clarity.

From the practical point of view, it seems appropriate to first implement a pair of two auxiliary internal methods, with the help of which we will be able to add a new internal block (dynamically allocate it and remember it) or remove the existing (last and no longer used) block (deallocate it and forget it).

When it comes to the dynamic allocation as such, we will not use the usual operators `new` and `delete` (or their array variants), because their use automatically and inevitably involves performing the allocation and calling the constructor, and calling the destructor and performing the deallocation, respectively. Instead, we will work with low-level allocation, within which both the aspects are decoupled from each other.

In particular, we will allocate a new inner block using the `void* ::operator new(size_t size)` function, the only parameter of which is the size of the required memory in the number of bytes. If the allocation succeeds, we get a valid pointer to the beginning of the allocated space and just retype it from `void*` to `T*`. If the allocation fails, the usual exception `std::bad_alloc` is thrown. If, on the other hand, we want to deallocate an existing inner block, we will use the `void ::operator delete(void* ptr)` function, where we just pass a pointer to the beginning of such a block. Both these functions can be found in the `<new>` library.

As part of adding a new element into the corresponding free slot in the currently last block, we need to call the corresponding element constructor. We will use the placement new operator for this. I.e., construct `new (target) T(...)`, where `target` is a pointer to the place where a given instance of the element should be created (the place itself has already been allocated), and `...` will be replaced with specific values to be passed as parameters to the intended element constructor, which in our case will either be its copy or move (stealing) constructor. When removing an element, on the other hand, we must explicitly call its destructor, which is easily done via `target->~T()`.

As we have already described, the function for adding elements, as well as for adding blocks, must guarantee atomic behavior in terms of the expected consistency. This means that either the intended operation can be executed completely successfully, or, in case of even a partial failure, we must compensate for the effect of the actions already carried out and inform about the overall failure via a suitable exception.

We have already discussed the possibility of failure when allocating a new block using the `::operator new` function. However, let us realize that the `push_back` operation, through which we insert a pointer to such a newly allocated block into the vector at the first level of the internal storage, may also fail. When inserting a new element, its constructor may fail, too, which is recognized by the fact that it throws some exception. And that exception may really be completely arbitrary. Let us also note that construction of a

new object (of any kind) is considered successful if and only if it runs successfully until the very end. If not, the compiler automatically takes care of destroying any of its possible data members or ancestors (if they have already been successfully initialized), and so we do not and actually must not invoke destruction of a given object as a whole by ourselves (since it has simply not been created at all).

To access individual elements stored in our flexible array, we will provide the following two pairs of methods with apparent meaning:

- `T& at(size_t index)`
- `const T& at(size_t index) const`
- `T& operator[](size_t index)`
- `const T& operator[](size_t index) const`

Another part of this assignment is handling of edge situations that should not occur at all if the users use our container correctly. More precisely, we will offer the possibility to activate kind of a debug mode, within which we will detect such situations and throw relevant exceptions. However, if this mode is not active, which we understand to be the default option for the production environment, we will not perform such checks at all (and the relevant code fragments will not even become part of the compiled application).

The technical solution is simple. The relevant code within the implementation of our flexible array is placed into conditionally compiled sections using the `#ifdef` and `#endif` directives (or other similar ones, if needed). If intended, the user then activates the debug mode via the `#define` directive. In particular, we will use a macro named `ARRAY_DEBUG_MODE` for this purpose, and we will handle the following situations:

- `std::out_of_range("Invalid index")`: if in any access method `at` or `[]` the caller requests an invalid element index; in the case of the `at` functions, we always throw such an exception (regardless of the debug mode); within the `[]` operators, on the contrary, in the debug mode only (the reason for this non-symmetric approach is to provide the same behavior as the standard containers offer)
- `std::invalid_argument("Empty array")`: if the caller tries to execute the `pop_back` method over an empty flexible array (in the debug mode only)

Both the standard exceptions can be found in the `<stdexcept>` library. Let us emphasize that the purpose of our conditional checks is not to solve the correctness of our flexible array implementation, but to offer the users of this flexible array better options for debugging their applications.

Last but not least, we also implement four special member functions, the copy and move constructors and assignment operators. Their default implementation generated by the compiler, similarly to the destructor, would not suit our needs. Not only inside the implementation of some of them, another global function `void swap(Array<T>& array_1, Array<T>& array_2) noexcept` will prove useful. Its task is to mutually swap two instances of flexible arrays.

- `Array(const Array& other)` [♦]: copy constructor
- `Array(Array&& other) noexcept`: move (stealing) constructor
- `Array& operator=(const Array& other)` [♦]: copy assignment operator
- `Array& operator=(Array&& other) noexcept`: move (stealing) assignment

In the case of the move constructor and move assignment operator specifically, we perform the stealing from the source flexible array in such a way that its resulting state will correspond to a valid completely empty array. At the same time, we will also add preventive protection to both of these functions for situations where we might attempt to assign an instance to itself.

Put all the code in a single header file called `Array.h`, which will be the only file expected to be submitted. The test course will again be under the control of a predefined `main` function.

The assignment focuses on the ability to design a custom simple container with a non-trivial layout of its internal storage for individual elements. In practical terms, this means working with low-level allocation using the `::operator new` and `::operator delete` functions, direct construction of objects (via the placement new operator) and their destruction (by manually invoking destructors), as well as designing custom copy and move constructors and assignment operators, throwing standard exceptions, and conditionally compiled code. The goal of the assignment also includes covering general principles related to the exception guarantee levels, resource stealing rules, implications of the Rule of Five, and implicitly generated functions.