NPRG041 – 2024/25 Winter – Labs MS – **Small Assignment C10**

Flexible Array I

Within this and the next assignment, we will propose an implementation of our own generic container, namely a flexible array. Its behavior will largely be based on the standard vector container, but we will deliberately design and program some of its aspects differently. Specifically, in this task, we first focus on the internal storage of such a flexible array and its basic functionality. In the next assignment, we will look at advanced constructors and assignment operators, as well as implement custom 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.

The flexible array class, let us simply call it `Array`, will have a single template parameter `Element`, which is 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.

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 straight-forward index arithmetic to access particular elements.

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 as such. We then specifically define the necessary internal mechanisms in such a way that we will always have just the number of allocated inner blocks that is absolutely necessary to store the current number of elements. Of course, other smarter strategies could be used as well, but we do not want to complicate the whole situation for us unnecessarily.

First of all, let us have a look at three constructors we will offer in our flexible array class:

- `Array(size_t block_size)`: creates an empty flexible array instance; the passed parameter indicates the requested block size; if not explicitly specified, we will assume a default size equal to 10
- `Array(size_t count, const Element& item)`: creates a new flexible array instance such that it will contain `count` copies of the provided sample element `item`; as for the block size in this case, we automatically assume the default value mentioned above
- `Array(std::initializer_list<Element> items)`: creates a new flexible array instance, into which we directly insert copies of elements from the provided initialization list; we again assume the same default block size

We must also not forget to implement a destructor `~Array() noexcept`. Its task will be to arrange correct destruction of the entire flexible array. In connection with the destructor, we will also offer a public method `void clear()`, which will empty the container and so remove all its elements. We will then offer the following three methods, which will become useful especially for potential debugging purposes:

- `inline size_t size() const`: returns the current size of the flexible array, i.e., the number of elements currently inserted into it
- `inline size_t capacity() const`: returns the current size of the allocated internal storage space, i.e., the number of elements that all the currently allocated internal blocks are capable of storing
- `void print(std::ostream& stream = std::cout) const`: prints the content of the entire flexible array, assuming the format `[1, 2, 3, 4, 5]`; in other words, we first print a pair of square brackets and place a list of individual elements inside it separated by commas and spaces; these elements themselves will be printed using their operator `<<`

To increase the comfort of our users, we will also add the corresponding global function for the `<<` operator for printing the entire flexible array into an output stream.

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

- `void push_back(const Element& 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(Element&& 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` without copying; we become its owner again and the caller may no longer use the original element
- `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

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, first of all, 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).

As for dynamic allocation as such, we will not use the usual operators `new` and `delete` (or their variants for arrays), because their use automatically and inseparably involves performing the allocation and calling the constructor, and calling the destructor and performing the deallocation, respectively. Instead, we will work with low-level dynamic allocation, within which both the aspects can be separated from each other.

Therefore, we allocate a new inner block using the function `void* std::malloc(size_t size)`, 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 `Element*`. If the allocation fails, we get `nullptr`. If, on the other hand, we want to deallocate an existing inner block, we will use the function `void std::free(void* ptr)`, where we just pass a pointer to the beginning of such a block.

As part of adding a new element into the corresponding free slot in the currently last block, we need to call the corresponding constructor of a given element. We will use the placement new operator for this. Specifically, we will use construct `new (target) Element(...)`, 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 of 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->~Element()`.

We must be very careful when designing especially the following functions: the initialization constructor, repetition constructor, function for adding a new inner block, and (both) functions for adding a new element. Under all the circumstances, we must ensure that their behavior will be atomic as for the necessary consistency expectations. This means that either the intended operation can be executed completely successfully, or, in the case of 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.

In particular, we will throw the standard `std::bad_alloc` exception when an internal block allocation fails using the `std::malloc` function. Note also that the `push_back` operation, via which we insert the pointer to such a newly allocated block into the vector at the first level of the internal storage can also fail. When inserting a new element, its constructor may fail, too, which can be recognized by catching exceptions thrown by it. And these can really be absolutely arbitrary exceptions. Let us also note that the 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 itself takes care of destroying its possible non-trivial data members, as well as any ancestors in the inheritance hierarchy. We need to be aware of this fact while working on our initialization and repetition constructors, too.

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

- `Element& at(size_t index)`

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

The last 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 code).

The technical solution is, in fact, simple: if we are interested, we first activate this debug mode from the source file using the `#define __DEBUG__` directive. Then, in the header file with the implementation of our flexible array, we just use conditional sections in the form of 1) `#ifdef __DEBUG__` to open a conditional section, 2) the actual fragment of our code, and 3) `#endif` to end the section. We will, in particular, 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 `<exception>` 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.

Put all the code in a single header file called `Array.h`, which is the only expected file to be submitted. The test course will again be under the control of a predefined `main` function. The task focuses on the ability to design a custom simple container with a non-trivial organization of its internal storage for individual elements, work with initialization lists, low-level dynamic allocation and direct construction, and destruction of objects.