1 Introduction

In this assignment you will implement your own simplified version of `std::deque`. In order to this you will have to use and know the following concepts:

- class templates,
- non-type template parameters,
- pointers,
- special member functions,
- move-semantics,
- memory allocation for arrays (`new[]` and `delete[]`), more information further down,
- exceptions,
- sequential containers.

2 Description

We covered the inner workings of `std::deque` during the seminars (seminar 7). For details on how to implement it please review the slides.

In this assignment you should create a class-template called `deque` that takes two template parameters: `T`; the type of values stored in the deque, and `chunk_size`; how many values can be stored at most in each chunk. `deque` should have the following member functions:

- default constructor.
- all special member functions.
- `push_front` - add a value to the front of the deque.
- `push_back` - add a value to the back of the deque.
- `pop_front` - remove the first value in the deque.
- `pop_back` - remove the last value in the deque.
- `operator[]` - access the value at a specified index.
- `at` - access the value at a specified index, throws an exception if the index is out of range.

`at` and `operator[]` should have both a `const` and non-`const` version. It is important that both of these functions return references and not copies since we do not know how expensive copying these types are and we would rather spare us the headache of having to think about it.

You are to implement your deque in files `deque.h` (declaration) and `deque.tcc` (definitions). There are testcases given in `deque.cc`, expand these to test all functionality of your deque.

Non-functional requirements

Your deque must implement the following requirements:

- Once a value is added to the deque it should never move in memory again (unless we of course remove the value).
- There should be no limitation on how many value can be added to the deque.
- You are not allowed to use any container (except C-arrays and `std::array`) in your implementation.
- There should be no memory leaks.
- Lookup must be constant time (i.e. the number of values should not make the lookup functions slower).
- Insertion should be constant time in *most* cases (the exception is if the chunk array is full, then it may be linear time, but this should happen rarely). This also called *amortized constant time*.
- Copying the deque should result in a deep copy.

Note: We do not care about exception safety, nor do we care about calling constructors and destructors for individual elements when they are pushed or popped.

Hint: Only the inserted values have to be fixed in memory, the chunk array does not. Therefore, if the chunk array becomes full we can double its size by reallocating the array.

3 Array allocation

There is an alternative version of `new` called `new[]`. `new` is used to allocate memory for *one* value. `new[]` allocates one or *more* values placed sequentially in memory (an array). The syntax works as follows:

```
T* array = new T[N] {};
```

Here `T` is the data type of the values in the array we allocate, `N` is a variable that indicates the size (Note: it can be decided at run-time) of the array and `{}` indicates that each value should be default-initialized.

Notice that `array` is a pointer, not an array-type. `array` points to the first value in the array. It is your responsibility to remember how long the array is, so be sure save the value of `N`.

You can access values as if `array` was an actual array-type, i.e. like this `array[0] = 5`.

We can then deallocate this array with `delete[]`, as such:

```
delete[] array;
```