Dynamic Array

Implement your own version of the dynamic array `Dynarray` class. This is almost the same as the example in Lecture 17.

Expected Usage

A `Dynarray` represents an "array" of `int`s, the length of which is determined at run-time and the elements are stored in dynamically allocated memory. A `Dynarray` should manage the memory it uses correctly: It should allocate adequate memory when initialized, and deallocate the memory when it is destroyed to avoid memory leaks.

Initialization

Let `n` be a non-negative integer and `x` be an integer. Let `begin` and `end` be of type `const int *` satisfying `begin <= end`, with `begin` pointing at the first element of some array and `end` pointing at **the element after** the last element of that array. A `Dynarray` can be constructed in the following five possible ways:

- ```
  Dynarray a;
  ```

  Initializes the object `a` to be an empty "array" whose length is `0`. (Default-initialziation)

- ```
  Dynarray a(n);
  ```

  Initializes the object `a` to be an "array" of length `n`, all elements **value-initialized**.

  Note: A constructor with one parameter also defines a **type conversion**. For example, the `std::string` class has a constructor that accepts one argument of type `const char *`, so the implicit conversion from C-style strings to C++ `std::string`s is supported. However, we definitely don't want to see the abuse of such constructors:

  ```
  Dynarray a = 42;
  ```

  or

  ```
  void fun(Dynarray a);
  fun(42); // constructs Dynarray(42) and passes it to `a`
  ```

  In order to forbid the use of this constructor as an implicit type conversion, you should add the `explicit` keyword:

  ```
  class Dynarray {
    // ...
    explicit Dynarray(std::size_t) /* ... */
  };
  ```

- ```
  Dynarray a(n, x);
  ```
  ```

Initializes the object `a` to be an "array" of length `n`, all elements initialized to be `x`.

- ```
  Dynarray a(begin, end);
  ```

  Initializes the object `a` to be an "array" of length `end - begin`. The elements are obtained from the range `[begin, end)`. For example,

  ```
  const int arr[10] = {19, 64, 10, 16, 67, 6, 17, 86, 7, 29};
  Dynarray a(arr + 3, arr + 7); // a contains the elements {16, 67, 6, 17}
  ```

- ```
  Dynarray b = a;
  ```

  Copy-initialization. See below.

Copy control

Let `a` be some existing `Dynarray` object. The following are equivalent ways of initializing a new `Dynarray` object:

```
Dynarray b = a;
Dynarray b(a);
Dynarray b{a};
```

This is the copy-initialization from `a`. Our `Dynarray` adopts the **value semantics**: Such copy-initialization should allocate another block of memory for `b` and copy the elements from `a`. After that, the data (elements) owned by `a` and `b` should be independent: Modification to some element in `a` should not influence the elements in `b`.

If `b` is also an existing `Dynarray` object, the following assignment can be performed:

```
b = a;
```

This is the copy-assignment from `a`. After that, `b` should be a copy of `a`. Any modification to an element of `a` should not influence the elements in `b`.

Your copy-assignment operator must be self-assignment safe.

Basic information

Let `a` be an object of type `const Dynarray`. The following operations should be supported:

- ```
  a.size()
  ```

  Returns the length of the "array", that is, the number of elements in `a`. It should be of type `std::size_t`, which is defined in `<cstddef>`.

- ```
  a.empty()
  ```

  Returns a `bool` value indicating whether the `Dynarray` is empty or not. A `Dynarray` is said to be *empty* if its length is zero.

Element access

Let `a` be an object of type `Dynarray` and `ca` be an object of type `const Dynarray`. Let `n` be a non-negative integer. The following operations should be supported:

- ```
  a.at(n)
  ```

  Returns a **reference** to the element indexed `n` in `a`. It is both readable and modifiable since `a` is not `const`. For example:

  ```
  a.at(n) = 42;
  std::cout << a.at(n) << std::endl;
  ```

- ```
  ca.at(n)
  ```

  Returns a **reference-to-`const`** to the element indexed `n` in `ca`. It should be read-only, since `ca` is `const`. For example:

  ```
  std::cout << ca.at(n) << std::endl; // OK
  ca.at(n) = 42; // This should lead to a compile-error.
  ```

Moreover, to keep in consistent with the behaviors of the standard library containers, `Dynarray::at` should do bounds-checking. If `n` is not in the range `[0, a.size())`, you need to **throw an exception** `std::out_of_range`. To throw this exception, write

```
throw std::out_of_range{"Dynarray index out of range!"};
```

The exception class `std::out_of_range` is defined in the standard library file `<stdexcept>`.

Examples

Write your code in `dynarray.hpp`. We have provided a template for you to begin with, although it contains only some preprocessor directives.

We have also provided a `compile_test.cpp` for you which contains the compile-time checks of your implementation. Members missing, `const` qualifier missing, or incorrect return types will be detected.

Here is a sample usage of the `Dynarray`:

```
#include "dynarray.hpp"
#include <algorithm>
#include <iostream>

void reverse(Dynarray &a) {
  for (int i = 0, j = a.size() - 1; i < j; ++i, --j)
    std::swap(a.at(i), a.at(j));
}

void print(const Dynarray &a) {
  std::cout << '[';
  if (!a.empty()) {
```

```cpp
    for (std::size_t i = 0; i + 1 < a.size(); ++i)
      std::cout << a.at(i) << ", ";
    std::cout << a.at(a.size() - 1);
  }
  std::cout << ']' << std::endl;
}

int main() {
  int n;
  std::cin >> n;
  Dynarray arr(n);
  for (int i = 0; i != n; ++i)
    std::cin >> arr.at(i);
  reverse(arr);
  print(arr);
  Dynarray copy = arr;
  copy.at(0) = 42;
  std::cout << arr.at(0) << '\n'
            << copy.at(0) << std::endl;
  return 0;
}
```

Input:

```
5
1 2 3 4 5
```

Output:

```
[5, 4, 3, 2, 1]
5
42
```

Submission

Submit the contents of your `dynarray.hpp` to the OJ.