Dynamic Array 2.0

This problem is based on the `Dynarray` you wrote in Homework 5 Problem 3. Before adding anything new to it, your `Dynarray` should meet all the requirements in that problem first.

In this task, your `Dynarray` should support the following new things.

1. Type alias members

The standard library containers have many type alias members for the purpose of supporting generic types. For example, `std::vector<T>::value_type` is `T`, `std::vector<T>::size_type` is `std::size_t`, `std::vector<T>::pointer` is `T *`.

As an exercise, do the same thing in your `Dynarray`. You need to define the following type alias members:

type alias member	definition
`Dynarray::size_type`	`std::size_t`
`Dynarray::value_type`	`int`
`Dynarray::pointer`	`int *`
`Dynarray::reference`	`int &`
`Dynarray::const_pointer`	`const int *`
`Dynarray::const_reference`	`const int &`

All of them should be `public`.

Moreover, we will eventually make this `Dynarray` a class template `Dynarray<T>` that can store any types of data, not only `int`s. This will be in Homework 7 or 8, depending on the lecture schedule. To make your work easier by then, you'd better make full use of the type alias members you have defined. For example, change `new int[n]` to `new value_type[n]`, change `int &` to `reference`, and change `(const int *begin, const int *end)` to `(const_pointer begin, const_pointer end)`, etc. By the time we make this a class template, you will just have to modify very few things.

2. Move operations

Add a move constructor and a move assignment operator for your `Dynarray`. The move operations of `Dynarray` have the semantics of transferring the ownership of the data. For example, suppose `a` is a `Dynarray`, and the move-initialization of `b`

```
Dynarray b(std::move(a));
```

makes the ownership of `a`'s data transferred to `b`. After that, `a` should be an empty dynamic array that can be safely destroyed or assigned to. The move assignment operator has similar semantics.

Both the move constructor and the move assignment operator should be `noexcept`, and they should not involve any operations that might throw exceptions (e.g. `new`/`new[]` expressions).

3. Find

Let `a` be a `Dynarray` and `x` be an `int`. `a.find(x)` should return the position (index) where `x` first appears. For example:

```
int arr[] = {42, 43, 45, 43, 45, 47, 42};
Dynarray a(arr, arr + 7);
assert(a.find(45) == 2);
assert(a.find(43) == 1);
assert(a.find(47) == 5);
```

If `x` is not found, return `Dynarray::npos`, which should be of type `const std::size_t` and has the value equal to `static_cast<std::size_t>(-1)`.

Moreover, one can also specify where to start searching by passing the second argument `pos` of type `std::size_t`. For example:

```
assert(a.find(43, 2) == 3);
assert(a.find(42, 1) == 6);
assert(a.find(43, 4) == Dynarray::npos);
assert(a.find(42, 19260817) == Dynarray::npos);
```

`a.find(x)` is equivalent to `a.find(x, 0)`. If `x` is not found in the index range `[pos, size())`, or if `pos >= size()`, `Dynarray::npos` should be returned.

Notes:

- The return type of `find` should be `std::size_t`.
- C++ allows functions to have default arguments.
- `const` static members can have an in-class initializer.

Submission

We have provided you with three files: `dynarray.hpp`, `example.cpp` and `compile_test.cpp`. You can run `example.cpp` and `compile_test.cpp` on your own. Submit the content of `dynarray.hpp` to OJ.

Grading

The grading of this problem contains three subtasks.

The first subtask contains only compile-time checks. This subtask accounts for 5 points.

The second subtask contains the tests from Homework 5 Problem 3, which accounts for 5 points.

The third subtask contains the new tests for the requirements in this problem, which accounts for 90 points.

If subtask 2 is not passed, the testcases for subtask 3 will not be run.