Problem 1. Dynamic Array 3.0

终于，是时候要把 `Dynarray` 变成一个可以存储各种可能的类型的元素的类模板 `Dynarray<T>` 了！

对元素类型 `T` 的要求

首先，不是什么类型的元素都能往 `Dynarray` 里存的。我们需要注意的情况主要有以下两点：

1. `T` 不能是一个引用。引用是十分特别的类型，它并不是一个实际的对象，不能默认初始化，而且对于引用的初始化和赋值的含义也有根本上的区别。因此，我们应当拒绝实例化任何的 `Dynarray<T &>` 和 `Dynarray<T &&>` 。如果用户希望在 `Dynarray` 中存储“引用”，可以使用指针代替，或者存储 `std::reference_wrapper<T>` （它其实也是一个对指针的封装）。
2. `T` 不能是 `const` 或 `volatile` （简称 cv-qualified ） 的。你可能不知道什么是 `volatile` ，没有关系，只需记住它是一种和 `const` 用法类似的特殊 qualification ，例如我们有“指向 `volatile` 的指针”和“`volatile` 指针”，有“绑定到 `volatile` 的引用”，有“`volatile` 成员函数”，有禁止去除底层 `volatile` 的规则等等。 `T` 之所以不能带有 cv-qualification ，是因为我们的 `Dynarray` 具有**值语义**（ value semantics ）：一个 `const Dynarray<T>` 就是一个存储 `T` 类型的元素但禁止修改的 `Dynarray` ， `Dynarray<T>` 本身不是 `const` 就意味着它的元素可以修改。

标准库的许多容器（例如 `std::vector` ） 对于其存储的元素的类型也有以上要求。为了体现这两点要求，我们可以在 `Dynarray` 的开头使用 `static_assert` ：

```
template <typename T>
class Dynarray {
  static_assert(/* condition */, /* message */);
  static_assert(/* condition */, /* message */);
  // ...
};
```

其中 `/* condition */` 必须是一个编译时已知的 `bool` 表达式， `/* message */` 必须是一个字符串字面值。 `static_assert(condition, message)` 会在编译时检测 `condition` 是否成立，如果不成立则报告一个编译错误，并且报错信息中会含有 `message` 的内容。**请编写有意义的 `message` ！**

标准库文件 `<type_traits>` 中定义了 `std::is_const_v<T>`, `std::is_volatile_v<T>`, `std::remove_cv_t<T>`, `std::is_same_v<T, U>`, `std::is_reference_v<T>` 等等这些关于类型的“元函数”，其中，以 `_v` 结尾的都是编译期 `bool` 常量，以 `_t` 结尾的都是类型别名。你可能需要使用它们。在这里可以看到它们的文档和示例。

类型别名成员

在上一次作业中，我们已经为 `Dynarray` 定义了一些类型别名成员。这次需要进行的改动如下：

- `value_type`, `pointer`, `reference`, `const_pointer`, `const_reference` 都应当是关于 `T` 的，而非关于 `int` 的。
- `Dynarray<T>::iterator` 和 `Dynarray<T>::const_iterator` 应该是 `Dynarray<T>` 的两个正向迭代器类型。由于 `Dynarray<T>` 的存储方式十分简单，迭代器完全可以使用指针代替。
- `Dynarray<T>::difference_type` 是两个 `Dynarray<T>::iterator` 相减的结果类型。
- `Dynarray<T>::reverse_iterator` 和 `Dynarray<T>::const_reverse_iterator` 是 `Dynarray<T>` 的两个反向迭代器类型。对于一个普通的（正向）迭代器 `Iter` ，其对应的反向迭代器就是 `std::reverse_iterator<Iter>` 。

成员函数

在上一次作业中，我们建议你将所有涉及到元素类型的地方都使用前面声明的类型别名成员来代替，例如将 `new int[n]` 替换为 `new value_type[n]` ，将 `const int &` 替换为 `const_reference` 。如果你已经这么做了，那么除了以下三个改动之外，你什么都不用改。如果没有，那么你还需要仔细检查各个函数，将其中假定元素类型为 `int` 的部分改掉。

1. 将接受一对指针的构造函数 `Dynarray::Dynarray(const int *, const int *)` 改为接受一对 **ForwardIterator** 。用户究竟会传进何种迭代器类型我们不得而知，所以我们直接以模板的方式接受参数：

   ```
   template <typename ForwardIter>
   Dynarray(ForwardIter begin, ForwardIter end);
   ```

   但是注意，并不是你给它命名为 `ForwardIter` 它就真的是 ForwardIterator 了。我们还需要限定它的迭代器型别：取出 `typename std::iterator_traits<ForwardIter>::iterator_category` 然后判断它是不是 `std::forward_iterator_tag` 或其子类，如果不是的话就不能允许实例化这个函数。这里我们借助 **SFINAE** （Substitution Failure Is Not An Error）手法实现：

   ```
   template <typename ForwardIter,
             typename = std::enable_if_t<std::is_base_of_v<
                 std::forward_iterator_tag,
                 typename std::iterator_traits<ForwardIter>::iterator_category>>>
   Dynarray(ForwardIter begin, ForwardIter end);
   ```

   ForwardIterator 不一定具有 `operator-` ，所以你不能使用 `end - begin` 来获取这两个迭代器的距离，而应该使用 `std::distance` 函数。

   有了这个构造函数，我们可以使用任意的一对迭代器来初始化一个 `Dynarray<T>` ，只要那个迭代器所指向的元素类型能用来给一个 `T` 类型的对象赋值即可。例如

   ```
   std::vector vec{2, 3, 5, 7};
   Dynarray<int> di1(vec.begin() + 2, vec.end()); // {5, 7}
   std::list lst{2.5, 3.7, 5.0, 7.2};
   Dynarray<int> di2(lst.begin(), lst.end()); // Truncated towards zero, {2, 3, 5, 7}
   Dynarray<int> di3(di2.crbegin(), di2.crend()); // {7, 5, 3, 2}
                                                  // 使用了 Dynarray 的反向迭代器，见下
   ```

   要写这么多 `<int>` 真是麻烦，这里的 `di1` 和 `di3` 的元素类型实际上都等于传入的迭代器所指向的元素类型。有没有办法让编译器自动根据 `ForwardIter` 指向的元素类型来推导出 `T` 呢？当然，只需**在类外**提供一个 deduction guide ：

   ```
   template <typename ForwardIter>
   Dynarray(ForwardIter, ForwardIter)
       -> Dynarray<typename std::iterator_traits<ForwardIter>::value_type>;
   ```

   有了这个之后，上面的代码中的 `di1` 和 `di3` 前面的 `<int>` 都可以不用写了。

2. 迭代器的获取： `begin()` , `end()` , `cbegin()` , `cend()` , `rbegin()` , `rend()` , `crbegin()` , `crend()` 。其中

   - `c` 表示 "const" ， `r` 表示 "reverse" 。 `cbegin()` , `cend()` , `crbegin()` , `crend()` 都只能在 `const` 对象上调用，前两个返回 `const_iterator` ，后两个返回 `const_reverse_iterator` 。

- `begin()`, `end()`, `rbegin()`, `rend()` 都应该具有 `const` 和 non-`const` 的重载，其中 `const` 版本返回 `const_iterator` 和 `const_reverse_iterator`， non-`const` 版本返回 `iterator` 和 `reverse_iterator` 。
- 你可以参考标准库 `std::vector` 的这 12 个函数的定义，特别是如何创建 `reverse_iterator` 和 `const_reverse_iterator` 。你只需要在 VSCode 中写一段这样的代码：

  ```
  std::vector<int> vec;
  auto rit = vec.rbegin();
  ```

  然后按住 ctrl ，鼠标左键点击 `rbegin` 就可以看到标准库 `std::vector` 的这一串成员函数的定义。其中一切有关 "nodiscard", "constexpr" 和 `noexcept` 的东西都可以忽略。

3. 正确编写适用于模板类的六个关系运算符 `<`, `<=`, `>`, `>=`, `==`, `!=` 以及输出运算符 `<<` 。不同的人在此处需要做的改动可能不同，例如有的人需要给每个函数都加上 `template <typename T>` 并给 `Dynarray` 加上 `<T>` ，而那些将它们都作为 `friend` 并且在类内定义的人几乎什么都不用改。

特别说明：模板代码中的每一个函数都应当将对于未知的模板类型参数 `T` 的要求降到最低，例如

- 接受一个长度参数 `n` 的构造函数应将 `n` 个元素进行**值初始化**，而不是将它们初始化为零。这个函数只能假定 `T` 是 default-initializable 的（即可以进行默认初始化和值初始化），不能依赖于 `T` 的任何拷贝或移动操作。
- 接受一个长度参数 `n` 和一个值 `x` 的构造函数需以 `const value_type &` 的方式传递 `x` ，并且只能假定 `T` 是 default-initializable 且可以进行拷贝赋值的。
- 拷贝操作可以假定 `T` 可以拷贝构造、可以拷贝赋值。
- 移动操作不能对 `T` 所支持的操作作任何假定。 `T` 甚至不需要是可以移动的。
- `==` 和 `!=` 只能依赖 `T` 的相等运算符 `==` ；四个比较运算符 `<`, `<=`, `>`, `>=` 只能依赖 `T` 的小于运算符 `<` 。
- `<<` 可以使用 `os << x` 输出一个 `T` 类型的对象 `x` ，其中 `os` 是一个输出流对象。

一份正常的、直接的实现应当很自然地满足这些要求。