Problem 2. Dimensional analysis

请去 Piazza Resources 下载所需的文件。

国际标准量纲制规定了物理量的基本量纲为：质量、长度或位置、时间、电荷、温度、密度以及物质的量，其它量纲则在此基础上复合而成。一个复合量纲可以看作若干基本量纲的幂的乘积。

我们在课上展示了一种技术，能够为每一种量纲定义一个独一无二的类型，并正确地定义各个量纲之间的运算，从而让含有量纲错误的计算程序无法通过编译。在本题中，你的任务是设计、实现这个思想，并编写一小段测试代码来验证你的实现的正确性。

代码框架

我们从这样两个类入手：

```
template <int Mass, int Length, int Time, int ElectricCurrent, int Temperature,
          int LuminousIntensity, int AmountOfSubstance>
struct Dimensions {};

template <typename T, typename Dim> class Quantity {
  T mValue;

public:
  constexpr explicit Quantity(const T &x) : mValue{x} {}

  constexpr const T &value() const { return mValue; }
};
```

`Dimensions<...>` 是一个类模板，它有七个 `int` 模板参数分别表示各个基本量纲的幂次，它所表示的量纲就是这些基本量纲的相应幂次的乘积。注意，这个类不需要任何成员，它的作用就是为每一个七元组定义一个独一无二的类型，这样的空类型在 C++ 里同样可以携带许多重要的信息。相比之下， C 的 `struct` 能做的事情就少了很多，在 C 中定义一个没有任何成员的 `struct` 被认为是 undefined behavior 。

`Quantity<T, Dim>` 是一个类模板，其中 `T` 是一个类型，表示这个量的类型，它通常会是某些浮点数类型，但也有可能是复数 `std::complex` 等； `Dim` 是一个 `Dimensions` 的特化，表示这个量的量纲。下面的代码创建了一个表示质量的浮点数变量：

```
Quantity<double, Dimensions<1, 0, 0, 0, 0, 0, 0>> m(42);
```

注意，我们目前没有引入任何单位信息，一个合理的设计是这些量的单位都是国际标准单位，例如质量的单位默认为 kg 。你也可以思考一下如何在这套系统中引入单位。

`Quantity` 的构造函数和 `value()` 函数都被声明为 `constexpr` ，这是因为 `Quantity` 的运行时属性实在是非常简单——它其实就是一个算术类型套了层壳而已，它的相关操作也基本以计算为主，所以它的几乎所有操作都可以在编译时计算。我们建议你为 `Quantity` 有关的所有函数都标上 `constexpr` ，以让编译器的能力发挥到最大。

在下发的代码中，我们使用了这样的方式来确保 `Quantity<T, Dim>` 中的 `Dim` 一定是 `Dimensions` 的一个特化

```
namespace detail {

template <typename T> inline constexpr auto isDimensions = false;

template <int M, int L, int T, int EC, int Tmp, int LI, int AOS>
inline constexpr auto isDimensions<Dimensions<M, L, T, EC, Tmp, LI, AOS>> =
  true;

} // namespace detail

template <typename T, typename Dim> class Quantity {
  static_assert(detail::isDimensions<Dim>);
  // ...
};
```

如果你不慎给 `Quantity` 的第二个模板参数传了些别的东西，编译器将会报告 static assertion failed（如果你不知道什么是 `static_assert`，去看第一题）。这里我们用一个变量模板 `isDimensions` 来判断一个类型是不是 `Dimensions<...>`，并将它藏在了 `namespace detail` 里。这是一个十分常见的做法：将那些不必暴露给用户看的实现细节藏进 `namespace detail` 里，有助于减少名字空间污染。你也可以编写一些辅助类或辅助函数，然后将它们藏在这个 `namespace` 里面。

我们还可以为七个基本量纲以及标量（`Scalar`）提供一份别名声明：

```
template <typename T>
using Scalar = Quantity<T, Dimensions<0, 0, 0, 0, 0, 0, 0>>;
template <typename T> using Mass = Quantity<T, Dimensions<1, 0, 0, 0, 0, 0, 0>>;
template <typename T>
using Length = Quantity<T, Dimensions<0, 1, 0, 0, 0, 0, 0>>;
template <typename T> using Time = Quantity<T, Dimensions<0, 0, 1, 0, 0, 0, 0>>;
template <typename T>
using ElectricCurrent = Quantity<T, Dimensions<0, 0, 0, 1, 0, 0, 0>>;
template <typename T>
using Temperature = Quantity<T, Dimensions<0, 0, 0, 0, 1, 0, 0>>;
template <typename T>
using LuminousIntensity = Quantity<T, Dimensions<0, 0, 0, 0, 0, 1, 0>>;
template <typename T>
using AmountOfSubstance = Quantity<T, Dimensions<0, 0, 0, 0, 0, 0, 1>>;
```

在正确地定义了算术运算（见下）的情况下，我们希望可以这样使用以上这些类：

```
template <typename T>
using Acceleration = Quantity<T, Dimensions<0, 1, -2, 0, 0, 0, 0>>;

int main() {
  Acceleration<double> g(9.8);
  Time<double> t(42.0);
  auto h = Scalar<double>(0.5) * g * pow<2>(t);  // Length<double>
  std::cout << h.value() << std::endl;           // 8643.6
  auto t_2 = sqrt(Scalar<double>(2.0) * h / g);  // Time<double>
  std::cout << t_2.value() << std::endl;         // 42
  return 0;
}
```

这个程序描述了一个物体在忽略空气阻力的情况下作自由落体运动 42 秒的运动情况。

以上代码中的 `<double>` 在 C++20 以前仍然需要手动写出，因为 `Acceleration`, `Time`, `Scalar` 这些都是 alias template 而非直接的类模板，而 alias template argument deduction 要到 C++20 才被支持。如果你嫌 `Scalar<double>(x)` 太麻烦，你也可以写一个辅助函数 `scalar(x)`

```
template <typename T>
constexpr Scalar<T> scalar(const T &x) {
  return Scalar<T>(x);
}
```

不过这个函数的名字不是很好，它和 `Scalar` 这个类型别名模板过于雷同；如果起名为 `makeScalar` 又显得很啰嗦。我们也可以退而求其次，至少为字面值提供一种快速的转换为 `Scalar` 的方式：

```
constexpr Scalar<double> operator""_scalar(long double x) {
  return Scalar<double>(static_cast<double>(x));
}
```

这样我们就可以写 `0.5_scalar` 来获得一个 `Scalar<double>(0.5)` 了。

算术运算

接下来的任务就交给你了。你需要为 `Quantity` 定义一元正号 `+x` 、一元负号 `-x` 、复合赋值运算符 `+=`, `-=`, `*=`, `/=` ，二元算术运算符 `+`, `-`, `*`, `/` ，以及整数次幂 `pow<N>(x)` 和根号 `sqrt(x)` 。你可能需要仔细思考下面几个问题：

1. `*=` 和 `/=` 仅在右侧运算对象具有何种量纲的时候可以支持？
2. 如何表示 `Quantity<T, D1>` 和 `Quantity<T, D2>` 相乘/除的结果类型？它应当是 `Quantity<T, DR>` ，其中 `DR` 是 `D1` 和 `D2` 的每一维对应相加/减的结果。
3. 如何处理数据类型不同的情况？例如，一个 `Mass<double>` 和一个 `Mass<float>` 相加，应该报错，还是应该得到一个 `Mass<double>` ？如果你希望支持这样的操作，你可能需要借助标准库 `<type_traits>` 里的 `std::common_type` ，但它无法照顾到有复数 `std::complex` 参与的情况，这可能需要一点工作量。当然，你也可以拒绝支持这类操作。
4. 我们没有支持量纲的非整数次幂，那么 `sqrt(x)` 应当在什么情况下允许编译？你需要让非法的开根操作产生编译错误，这一点可以借助 `static_assert` 来完成。

本题的 OJ 测试只会检测你有没有定义这几个函数。

其它设计

思考以下问题，并按照你的想法实现。

目前 `Quantity` 有一个 `value()` 函数返回它所表示的量的数值，但没有提供任何直接修改该数值（即 `mValue`）的方法。你打算提供什么成员函数来允许修改这个数值吗？如果有，你打算怎么做？如果没有，为什么？

目前，普通的算术类型变量想要参与这些物理量的运算，需要套一层 `Scalar<...>()` ，例如 `Scalar<double>(2.0) * x` 而不是 `2.0 * x` 。你打算支持 `Quantity<T, D>` 和 `T` 的直接运算吗？如果是，怎么做？如果不，为什么？

如果要往这套系统里加入指定单位的功能，怎么做？有兴趣的话可以尝试实现。

测试

本题的 OJ 测试只会检测你有没有定义算术运算这一节中要求的几个函数。通过 OJ 测试，你将获得基础分 40 ，剩下的得分由你和 TA 的面对面（可以线上） check 来决定。

你需要编写至少一段测试代码并现场编译、运行，来说明你的实现的正确性。这些测试代码应该

- 位于单独的 `.cpp` 文件中，并 `#include "quantity.hpp"` ，而不是和前面的代码混在一起。
- 7 个基本量纲都应当有所涉及，以证明你对于 7 个模板参数的处理都是基本正确的。
- 对算术运算一节中要求的所有算术运算都有基本的测试。
- 对其它设计一节中你实现了的功能有基本的测试。