Assignment 7 - Weights

Submit Assignment

Due Nov 15 by 11:59pm **Points** 0 **Submitting** a file upload **File Types** hpp
Available Nov 6 at 12am - Nov 17 at 11:59pm 12 days

The purpose of this assignment is to give you an initial experience in developing C++ template code. Additionally you'll gain experience in implementing mathematical operators, both as part of a class and external to a class.

The subject of the assignment is the creating of a `weight` class that can be used to represent different weight measures. Rather than storing a weight as a `double` or `int` or something else, we want to represent a weight as strongly typed kilograms, ounces, or pounds, along with conversions between these types. In order to accomplish this, it is necessary to write template code to represent a weight type.

The weight class is patterned after the `std::chrono::duration` type. A weight is defined by a ratio relative to a gram, a count of the number of items of the type, and the data type used to store the count. For example, a gram has a ratio of 1:1, while a kilogram has a ratio of 1000:1. If we have a weight of 500 grams, then a gram type has a count of 500 and a kilogram has a count of 0.5 (when using a floating point storage type).

This assignment will challenge you to think carefully about writing generic code. My implementation of the weight class is less than 100 lines. You won't write a lot of code, instead, you'll be writing the "right" code.

Assignment

Write a templated `weight` class, contained with a `usu` namespace according to the following specifications.

- The type is templated on two parameters:
  - A ratio (`std::ratio`) relative to the number of grams.
  - The data type used to store the weight count. The default for the storage type must be `std::uint64_t`.
- A default constructor that initializes the storage count to 0.
- An overloaded constructor that accepts a count of the weight type. For example, if the type is grams and the count is 4, the weight represents 4 grams.
- A `count` method that returns the count of the weight type.
- Overload the `+` operator to add two items of the same type.
- Overload the `-` operator to subtract two items of the same type.
- Overload the `*` operator to multiply the type by a scalar (integral or floating point).
  - These will be defined external to the class; not members of the class.
  - You'll need two overloads to do this, one for each position the scalar and weight type can occur; A scalar times a weight and a weight times a scalar.

Write a templated `weight_cast` function (not a member of the `weight` class) that converts from one weight type to another. This function is patterned after the `std::chrono::duration_cast` function. This function accepts a single template parameter that is the type to convert to, whereas the function parameter is the weight variable to convert from.

Define type aliases inside the `usu` namespace, in your weight.hpp file for:

- `microgram`

- `gram`
- `kilogram`
- `ounce`
- `pound`
- `ton`

Example Usage & Unit Tests

An example main.cpp is provided that exercises some of the capabilities of the code you need to write. Additionally you are provided a set of unit tests that exercise the capabilities of the weight class.

The last section of this assignment description shows the output from running the main.cpp code over my implementation of the weight class.

Files

- **main.cpp**
- **TestWeight.cpp**

Template Notes

Consider the following declarations...

```
usu::weight<std::ratio<10,1>> decigram(10);
usu::weight<std::ratio<10,1>, double> decigram2(10.2);
```

Notice the use of `std::ratio`. This is found in the `<ratio>` standard library header file.

The first declaration uses the `weight` class to declare a weight that has a ratio of 10:1 grams. Meaning that for each 1 count, it represents 10 grams. The variable name is `decigram` and is initialized with a value of 10. Therefore, this weight represents 100 grams. Because there is no storage type in the weight declaration, the underlying storage is `std::uint64_t`.

The second declaration adds a storage type parameter, a `double`. This type overrides the default `std::uint64_t` and becomes the storage type for the count of items in the weight. The constructor for this weight is passed 10.2, which can be represented by the `double` storage type, therefore, the count for this variable is 10.2.

Submission Notes

- Turn in only the following file: **weight.hpp**
- Your code must compile without any warnings or compiler errors; against the provide main.cpp and TestWeight.cpp provided code. See syllabus regarding code that has compiler errors.
- Your code must adhere to the CS 3460 coding standard: **link**
- Your code must be formatted through the use of the following clang-format configuration file: **link**

Example Run

```
--- From micrograms ---
micrograms : 1000000
grams      : 1
lbs        : 0.0022046226
ounces     : 0.0352739619
tons       : 0.0000011023
```

```
--- From pounds ---
micrograms : 907184740000
grams      : 907184
lbs        : 2000.0000000000
ounces     : 32000.0000000000
tons       : 1.0000000000

--- Operator Overloading ---
(pound + pound) : 1.00 + 0.50 = 1.50 ==> grams: 680
(pound - pound) : 1.00 - 0.50 = 0.50 ==> grams: 226
(pound * scalar) : 1.00 * 2.2  = 2.20 ==> grams: 997
(scalar * pound) : 3.2 * 1.00 = 3.20 ==> grams: 1451
```