Fixed Point - Task

Phase 3: Let It Grow

Why Should Forex Traders Suffer?

While using our Price works fine for most markets, from space rockets to chewing gum, there might be some sectors that will not be able to enjoy it. For example, forex trades are made with smaller units than 1 cent: The Pip. Pips are 1/10,000 of a single currency unit - for example, 0.01 cent or 0.0001 Dollar.
Since our class template only allows two digits after the point, it is not possible to use if for such small amounts. How can we solve that?

Rewrite it for Four Digits

Yes, we *can* rewrite a copy of the whole class template that uses 4 digits rather than 2. But code duplication is never a good idea.

Pass it in the Ctor

We might try to pass the number of digits after the decimal point to the ctors.

Wait! Why do we suggest that for the number of digits, but did not suggest it for the type of the underlying data member?

This solution is possible, but it suffers from a few drawbacks:
First, it doubles the size of the object.

If we use type long for the number itself, and keep the number of digits in an unsigned char, will it really double its size?

Second, it requires more calculations in runtime.
Third, it requires more memory fetches in runtime.
Fourth, what will happen when two such objects, with different digits number, will be used in calculations? For example, what will happen here?

```
Price<short> radians(3, 1415, 4)  // The 3rd arg is for digits num
Price<short> degrees(180, 0, 2)  // Ditto
std::cout << radians + degrees;
```

Both variables are of the same type - Price<short>, which means that operator+() for that type will be invoked (assuming you defined it). What should the result be? We cannot keep the precision if we change *radians* to have 2 digits, but we will immediately overflow if we change *degrees* to have 4 digits.
We might decide to return a *Price<long>* instead, but this does not make sense for the simpler, trivial case where both operands have the same number of digits after the decimal point.

What can we do, then, to solve that issue?

Integer Template Arguments

Just the same way we can define class template or function template with type arguments, we can also define them with integer arguments.
Thus, for example, the following:

```
template<class T, unsigned int SIZE> class Price { … };
```

Defines the template class *Price* to depend on a type *T* and on the number *SIZE*.

Another possible definition is

```
template<typename T, unsigned int SIZE> class Price { … };
```

What is the difference between both definitions?

Now, we have everything solved.
We should no longer provide the number of digits in the ctor; it will just be as simple and elegant as

```
Price<long, 2> somePrice(749, 90);
Price<int, 4> usdInNis(3, 6917);
```

In addition, since *SIZE* is resolved during compilation and is not a variable, the object size is left unchanged, run-time fetches are not required, and some of the calculations might also take place during compilation stage instead of run-time.

What will happen when trying to add (or whatever other operation) on two operands with different number of digits?

Some Fine Tunings

Now that we can use any number of digits after the decimal point, it makes no more sense to call that type *Price*. Rename the class template to *FixedPoint*.

The natural default type in C++ is *int*, so we might assume that anyone without other specific needs just like to use *int*. Just like function arguments, template arguments

can get default values, so you can make the type *int* by default. However, the number of digits must be explicitly set, so we would not like it to get a default value. Change the class template so that the default type is *int* but the number of digits does not have a default.

Will the following compile?

```
template<class T = int, unsigned int SIZE> class FixedPoint { ... };
```

How should you fix it?
