With the definition in the previous example, we can provide a `toRadians` function template that performs at maximum runtime efficiency by avoiding needless type conversions during the computation:

```
template <typename T>
constexpr T toRadians(T degrees)
{
    return degrees * (pi<T> / T(180));
}
```

Reducing verbosity of type traits

A **type trait** is an empty type carrying compile-time information about one or more aspects of another type. The way in which type traits have been specified historically has been to define a class template having the trait name and a public **`static`** data member conventionally called `value`, which is initialized in the primary template to **`false`**. Then, for each type that wants to advertise that it has this trait, the header defining the trait is included, and the trait is specialized for that type, initializing `value` to **`true`**. We can achieve precisely this same usage pattern replacing a trait **`struct`** with a variable template whose name represents the type trait and whose type of variable itself is always **`bool`**. Preferring variable templates in this use case decreases the amount of **boilerplate code**, both at the point of definition and at the ~~call site~~.[1]

Consider, for example, a Boolean trait designating whether a particular type `T` can be serialized to JSON:

```
// isSerializableToJson.h:

template <typename T>
constexpr bool isSerializableToJson = false;
```

The header above contains the general variable template trait that, by default, concludes that a given type is not serializable to JSON. Next we consider the streaming utility itself:

[1] As of C++17, the Standard Library provides a more convenient way of inspecting the result of a type trait, by introducing variable templates named the same way as the corresponding traits but with an additional `_v` suffix:

```
// C++11/14
bool dc1 = std::is_default_constructible<T>::value;

// C++17
bool dc2 = std::is_default_constructible_v<T>;
```

This delay is a consequence of the train release model of the Standard: Thoughtful application of the new feature throughout the vast Standard Library required significant effort that could not be completed before the next release date for the Standard and thus was delayed until C++17.