where it would be convenient to know whether the literal value is being negated. For example, if temperatures are being stored as **double** values in Kelvin and if the UDL suffix `_C` converts a floating-point literal from Celsius to Kelvin by calling a function, `cToK(double)`, then the expression `-10.0_C` produces the nonsensical value `-283.15` (`-cToK(10.0)`) rather than the intuitive value of `+263.15` (`cToK(-10.0)`). Alas, parsing the `-` sign as part of the literal is simply not possible.

Parsing numbers is hard

Many of the benefits of raw UDL operators and UDL operator templates require parsing integer and/or floating-point values manually, in code, often using recursion. Getting this right is tedious at best. The Standard Library does not provide much support, especially for **constexpr** parsing.

See Also

- "**decltype**" (§1.1, p. 25) introduces a keyword often helpful for deducing the return type of a UDL operator template.
- "**nullptr**" (§1.1, p. 99) describes a keyword that unambiguously denotes the null pointer literal.
- "**auto** Variables" (§2.1, p. 195) shows how **type inference** can be used to declare a variable to hold the value of a UDL when the type of the UDL varies based on its contents.
- "**constexpr** Functions" (§2.1, p. 257) explains how most UDLs can be used as part of a constant expression.
- "Inheriting Ctors" (§2.1, p. 535) discusses a feature that allows wrapper types (or **strong typedefs**) to be constructible from the same arguments as the type they wrap.
- "Variadic Templates" (§2.1, p. 873) shows how templates can take an infinite number of parameters, which is required for implementing UDL operator templates.
- "**inline namespace**" (§3.1, p. 1055) describes a feature not recommended for UDL operators, yet the C++14 Standard Library puts UDL operators into **inline** namespaces.