TMI on UDL

Steve Dewhurst
Semantics Consulting, Inc.

Let's Start With a Kvetch

- The Google C++ Style Guide is not very uplifting on UDLs:
  - "Do not use user-defined literals."
  - "User-defined literals (UDLs) allow the creation of new syntactic forms that are unfamiliar even to experienced C++ programmers, such as `"Hello World"sv` as a shorthand for `std::string_view("Hello World")`. Existing notations are clearer, though less terse."
  - "Do not overload `operator""`, i.e. do not introduce user-defined literals. Do not use any such literals provided by others (including the standard library)."
- A respected C++ blogger says, "While user defined literals look very neat, they are not much more than syntactic sugar."
- Where would our semantics be without syntax?
- Type sinks, that's where!

Type Sinks

- Some types are used as “sinks” for different kinds of semantic information.
- This results in dangerous interfaces like this:

```
class Date {
public:
   constexpr Date(size_t m, size_t d, size_t y) noexcept:
       month_ (m), day_ (d), year_ (y) {}
   ~~~
};
~~~
constexpr Date crash (10, 24, 1929);    // OK
constexpr Date crasher (24, 10, 1929);  // oops!
```

Avoiding Type Sinks

- The problem is that the same type, `size_t`, is used to represent three different concepts: month, day, and year.

✓ *Use a distinct type for each concept.*

- Like this:

```
class Day {
public:
    explicit constexpr Day(size_t d) noexcept
        : day_ (d) {}
    constexpr operator size_t() const noexcept
        { return day_; }
private:
    size_t day_;
};
```

Safety and Flexibility

- The `Date` class can now distinguish among the concepts of day, month, and year and treat them accordingly.
- Now we have the moral choice between being totalitarian, forcing all programmers to be just like us, crushing freedom and creativity, with the false promise of "correctness"…

```
class Date {
public:
 constexpr Date(Year y, Month m, Day d) noexcept:
        month_ (m), day_ (d), year_ (y) {}
    ~~~
};
```

Safety and Flexibility

- …or we acting like good Democrats, encouraging diversity without sacrificing technical correctness:

```
class Date {
public:
    constexpr Date(Month m, Day d, Year y) noexcept:
        month_ (m), day_ (d), year_ (y) {} // Americans
    constexpr Date(Day d, Month m, Year y) noexcept:
        month_ (m), day_ (d), year_ (y) {} // Europeans
    constexpr Date(Year y, Month m, Day d) noexcept:
        month_ (m), day_ (d), year_ (y) {} // us
    constexpr Date(Month m, Year y, Day d) noexcept:
        month_ (m), day_ (d), year_ (y) {} // Martians
    ~~~
};
```

A “Vexing” Gotcha

- But getting rid of type sinks addresses only half the problem. Your types have to be “easy to use correctly and hard to use incorrectly”:

```
constexpr size_t m = 10, d = 24, y = 1929;
Date crash (Month(m), Day(d), Year(y));     // ???
```

- The compiler parses the declaration as:

```
Date crash(Month m, Day m, Year y);         // a function!
```

- Traditionally, you’d fix the declaration by adding parentheses:

```
Date crash((Month(d)), Day(m), Year(y));    // an object
```

- User-defined literals can be clearer…

User-Defined Literals

- An integer-suffix may be used to specify the precise type of an integer literal.

  ```
  39U // unsigned int, not int
  ```

- A ***user-defined literal*** is just a literal followed by a *ud-suffix*.

  ```
  123.45_abc
  ```

- The *ud-suffix* allows a user-defined interpretation to be applied to the literal.

Literal Operators

- A ***literal operator*** is a function whose name has the form:

  `operator ""` *identifier*

- The first character of *identifier* should be "_" (an underscore).
  - Identifiers without a leading underscore are reserved.
- You can define a literal operator for `Day` as:

```
constexpr Day
operator "" _day(unsigned long long d) : noexcept
    { return Day(static_cast<size_t>(d)); }
```

- `_day` is a *ud-suffix* that may be applied to an integer literal. The argument type must be unsigned long long.

“Vexing” Gotcha Solved

- Again, you can define a `Date` object using:

```
constexpr size_t m = 10, d = 24, y = 1929;
Date crash ((Month(m)), Day(d), Year(y));    // vexing
```

- You can also use braced initialization syntax:

```
Date crash {Month(m), Day(d), Year(y)};     // OK
```

- But using user-defined literals is clearer:

```
Date crash (10_month, 24_day, 1929_year);  // clearer
Date rover {2_month, 2019_year, 13_day};   // bye...
```

What About Negatives?

- What if you're having a bad day?

```
Day monday = -10_day; // error!
```

- The minus sign is not lexically part of the literal, even for predefined literals like `-10`.
- You'll have to provide the proper overloaded operator for the return value of the literal operator.

```
constexpr Day operator –(Day d) noexcept {
    return d; // don't worry, be happy...
}
```

Money!

- Let's represent money as a variable amount of a fixed currency:

```
enum class Currency : unsigned { CAD, EUR, JPY, USD };

template <Currency C>
class Money {
    explicit constexpr Money(double amt) noexcept;
    ~~~
private:
    double amt_;
};
~~~
constexpr Money<Currency::USD> dollars (12.34); // yuck.
```

Literal Money

- We can also define floating literals for our `Money` types:

```
constexpr Money<Currency::USD>
operator "" _USD(long double amt) noexcept
    { return Money<Currency::USD>(amt); }
```

- The same convenience obtains:

```
constexpr auto dollars = 12.34_USD;          // non-yuck.
```

- The argument type for a floating literal must be `long double`.

Choices, Choices…

- The availability of user-defined literals gives us (even) more possibilities when we declare something:

```
// old-fashioned
constexpr Money<Currency::USD> amt1(21.43);

// Explicitly-Typed Initializer idiom
constexpr auto amt2 = Money<Currency::USD>(0.01);

// my two cents...
constexpr auto amt3 = 0.02_USD;       // correct.
```

- The UDL is easier to write and read.

Kinds of User-Defined Literals

- User-defined literals come in several varieties: integer, floating, character, and string.
- The types of the literal operator arguments are mostly fixed:

```
T operator "" _a(long double);          // floating literals
T operator "" _b(unsigned long long); // integer literals
T operator "" _c(char-type);            // char literals
T operator "" _d(string, size_t);       // string literals
```

- *char-type* is `char` or another character type, and *string* is `const char *` or another string type.

```
'x'_c      // call operator ""_c('x')
"Hello!"_d // call operator ""_d("Hello!", 6)
```

Overloaded Literal Operators

- Here's a poorly-designed stock:

```
class Stock {
public:
    Stock(string_view ticker);
    Stock(unsigned long long numeric_cusip);
    Stock(char ticker);
    ~~~
};
```

- A `Stock` can be initialized with a character string ticker, an all-digit CUSIP (!), or a single-character ticker. Weird.

Overloaded Literal Operators

- We can accommodate all three miserable initializations by providing different kinds of literal operators for the same ud-suffix.

```
Stock operator "" _stock(char const *ticker, size_t n)
    { return Stock(string_view(ticker, n)); }

Stock operator "" _stock(unsigned long long cusip)
    { return Stock(unsigned(cusip)); }

Stock operator "" _stock(char ticker)
    { return Stock(ticker); }
```

Overloaded Literal Operators

- Overload resolution selects the correct literal operator.

```
auto baba = "baba"_stock;
auto ibm = 459200101_stock;
auto ford = 'F'_stock;
```

Overloaded Literal Operators

- Alternatively, the overloads can also produce different types of literals. Here are two examples from the standard library:

```
auto baba = "baba"s;               // std::string
auto ibm = 459200101s;             // std::chrono::seconds
```

- I disagree with the choice of `s` as the ud-suffix, as `string` and `seconds` would have been clearer. (Score one for Google.)

```
auto yum = "yum"string;            // no comment required
auto ibm = 459200101seconds;
```

✓ *Prefer to associate the ud-suffix closely with the type returned by the user-defined literal.*

Overloading for Numeric Literals

- Suppose you're trying to stave off the death of a pleasant but idiosyncratic system of measurement.

```
class Slug {
public:
    constexpr Slug(long double amt) : amt_(amt) {}
    ~~~
private:
    long double amt_;
};

constexpr Slug operator ""_slug(long double a) noexcept
    { return Slug(a); }
```

Slugging it Out

- This works well for floating slugs.

```
auto m1 = 12.3_slug;
```

- But it won't work for integral slugs.

```
auto m2 = 12_slug; // error! No such literal operator.
```

- An overload for integers will fix the problem.

```
constexpr Slug
operator ""_slug(unsigned long long a) noexcept
    { return Slug(static_cast<long double>(a)); }
```

Convenience?

- Of course, it's possible to go overboard:

```
constexpr Date
operator "" _date(unsigned long long d) noexcept {
    return Date(
        Year(d / 10000),
        Month(d % 10000 / 100),
        Day(d % 100)
    );
}
~~~
Date gd = 1929'10'24_date;      // good idea?
```

Values and Formats

- The `_date` literal operator receives the value of the integer literal but cannot require a particular format of the literal.

```
1929'10'24_date          // desired format
19291024_date            // suboptimal
0x1265B90_date           // dismal
0111455620_date          // quite frightening
0b000001'001001'100101'101110'010000_date // ouch.
```

- If we want to enforce a particular format on our user-defined integer and floating literals, the best option is often to use a raw literal operator, which receives the individual characters that make up the literal.
- The raw literal operator parses the literal.

Raw Literal Operators

- A raw literal operator has the form:

```
T operator "" _e(char const *);
```

- Alternatively, it's possible to define a *literal operator template* that uses a character pack.

```
template <char...> T operator ""_f();
```

- The other literal operators are often called "cooked" literal operators.

Raw Literal Operators

- A raw literal operator is selected if there is no better match to an integer or floating literal.

```
constexpr unsigned
operator ""_base3(char const *n) noexcept {
    // parse n as a base 3 integer...
}
~~~
01120_base3 // call operator ""_base3("1120")
```

- You may have at most one "raw" operator for a ud_suffix, either a raw literal operator or a literal operator template.

Ud-Suffix Details

- A user-defined ud_suffix must start with an underscore.
- Additionally, names that start with an underscore and capital letter are “reserved identifiers.”
- So, in C++11 a user-defined ud-suffix must start with an underscore, optionally followed by anything that would make the result a legal identifier.
- That’s right:

```
constexpr T
operator "" _(long double a) noexcept // don't.
    { return T(a); }
~~~
auto aT = 12.3_;                        // just say no.
auto anX = 1729_______;                 // "no."
```

Ud-Suffix Details

- A sequence of characters like _2345789 is a legal identifier.
- That's right:

```
constexpr unsigned long long
operator "" _42(unsigned long long a) noexcept // don't.
    { return 42; }
~~~
auto ltuae = 123456_42;                    // just don't do it.™
```

But Flexibility Is Nice

- For example, `std::bind` can be confusing.

```
cout << afunc(100, 10.0, 1) << endl;
auto afunc2 = bind(afunc, _3, _1, _2);
cout << afunc2(100, 10.0, 1) << endl;
```

- You could try injecting comments with the placeholders.

```
constexpr decltype(auto)
operator "" _1(char const *, size_t) noexcept
  { return _1; }

constexpr decltype(auto)
operator "" _2(char const *, size_t) noexcept
  { return _2; }
```

...but...

- Now we can comment the placeholders without a comment.

```
auto afunc3 = bind(afunc,
                   "second int first"_3,
                   "first int second"_1,
                   "float last"_2);
```

- This is not necessarily a recommendation, but it does illustrate the basic C++ language philosophy of providing *non-prescriptive* flexibility in the language...that is tamed by idiom and convention.

✓ *We'll come back to this...*

Ud-Suffix Details In C++14

- In C++14, a ud-suffix may start with an underscore and a capital letter if there is no space between `operator ""` and the ud-suffix.

```
Stock operator ""_Stock(unsigned long long id)
    { return Stock(unsigned(id)); }
~~~
auto ibm = 459200101_Stock;                       // OK!
```

- It is thankfully reserved to the implementation to use this feature with arbitrary identifiers, even keywords.

```
Auto operator ""auto(char const *make, size_t n); // no.
auto ford = "ford"auto;                           // no.
```

Keyword UD-Suffixes

- This feature is lightly-used at present.

```
using namespace std::complex_literals;
auto as = 23if;
```

- But it may be useful in the future.

```
auto a = "C++"continue;
auto b = "Java"switch;
```

Cheering Up Google

- This is actually a useful feature.
- UDLs often generate class types, and many coding standards require classes to start with an uppercase letter.
- While this may be confusing…

```
Date operator "" _date(unsigned long long d) noexcept;
~~~
auto then = 1986'09'23_date; // is it a date or a Date?
```

this is less confusing.

```
Date operator ""_Date(unsigned long long d) noexcept;
~~~
auto now = 2019'09'23_Date;  // it's a Date
```

Literal Operator Templates

- A literal operator template takes a template parameter pack rather than an argument.
- The parameter pack must be precisely `char...`
- Here's a trivial example that gives the length of the literal:

```
template <char... chars>
constexpr auto operator "" _len() noexcept {
    return sizeof...(chars);
}
~~~
cout << 123'456_len; // 7
```

Restrictions

- The syntax for a literal operator template is fixed:

```
template <char c, char... chars>                // error!
constexpr unsigned operator ""_len(char c); // error!
```

- A literal operator template applies only to integer and floating literals.

```
1776'07'02_len // OK, 10
6.02e23_len    // OK, 7
"Hello!"_len   // error!
```

Overloading

- A literal operator template can overload an integer or floating literal operator.

```
template <char... cs>
constexpr Date                                    // #1
operator ""_Date() noexcept { ~~~ }

constexpr Date                                    // #2
operator ""_Date(unsigned long long arg) noexcept { ~~~ }
```

- As with a deduction context, the non-template literal operator is preferred.

```
1967'03'19_Date                                   // matches #2
1967.03'19_Date                                   // matches #1
```

Overloading

- Either a literal operator template or a raw literal operator may serve as a "catchall," but not both.

```
template <char... cs>
constexpr Date                                  // #1
operator ""_Date() noexcept { ~~~ }

constexpr Date                                  // #3, error!
operator ""_Date(char const *arg) noexcept { ~~~ }
```

Why Raw Operators Matter: Syntax

- Other numeric user-defined literals receive the *value* of the argument.

```
constexpr Date
operator ""_Date(unsigned long long arg) noexcept;
~~~
2001'01'01_Date  // #1: OK, arg is 20010101
20010101_Date    // #2: same, harder to read
0xDeadBeef_Date  // #3: not so good, arg is 3735928559
```

Value vs. Syntax

- A literal operator template receives the *characters* that form the literal. (So does a raw literal operator.)

```
template <char... cs>
constexpr Date operator ""_Date() noexcept;
~~~
2001'01'01_Date  // #1: chars are 2001'01'01
20010101_Date    // #2: chars are 20010101
0xDeadBeef_Date  // #3: chars are 0xDeadBeef
```

- A literal operator template may be used to parse the numeric literal in order to require a particular syntax/format.
- We could write `_Date` to insist on the first format.

Why Raw Literal Operators Matter: Value

- Compilers limit the sizes of numeric literals.

```
auto big    = 18'446'744'073'709'551'615;      // OK
auto bigger = 18'446'744'073'709'551'616;      // not OK
```

- For cooked user-defined literals, this imposes a limit.

```
auto slugs = 18'446'744'073'709'551'616_Slug; // not OK
```

- This is particularly a problem for user-defined extended precision numeric types.

```
auto biggest =
0xCafeBabeDef1edD1ab011ca1D00d5c01dedD00dDecea5ed_uint512;
```

Circumventing Limits

- An integer literal operator can’t handle the required precision.

```
constexpr uint512_t
operator ""_uint512(unsigned long long value) noexcept {
    ~~~
}
```

- A literal operator template has no such restriction on the value of an argument (though there may be an imposed limit on the size of the character pack).

```
template <char... chars>
constexpr uint512_t operator ""_uint512() noexcept {
    ~~~
}
```

Raw Literal or Literal Operator Template?

- It is guaranteed that the characters in a literal operator template are compile time constants.
- It's possible that a raw literal operator will be passed a string literal that is not a compile time constant. (Though it's unlikely.)

```
string zip = "303156";
auto ketchikan = operator ""_octal(zip.c_str()); // weird
auto ketchikan = 303156_octal;                   // usual
```

- As constexpr functions evolved from their restricted definition in C++11 to the more flexible and capable augmented definitions in C++14 and C++17, and as standard library containers are increasingly constexpr, raw literal operators have become an increasingly attractive option.

One Practical Difference…

- The argument to a constexpr function might not be a compile time constant. No static assertions.

```
constexpr auto
operator ""_quotes(char const *n) noexcept {
    static_assert(n != nullptr);                // error!
    ~~~
}
```

- A character pack is a compile time constant. Static assertions.

```
template <char... chars>
constexpr auto operator ""_quotes() noexcept {
    static_assert(sizeof...(chars));            // OK.
    ~~~
}
```

Another Very Practical Difference

- A literal operator template is a template.
- A raw literal operator is not.
- There are a whole lot of programming techniques that apply to templates that don’t apply to non-templates.
- Like SFINAE…but it’s not easy.
- Ordinary SFINAE must stick to the immediate context of a template.
- Imposed limitations on a literal operator template add ***restrictions*** leaving ***few options*** for SFINAE application:

```
template <char... chars>
constexpr enable_if_t<cond, Date>
operator ""_Date() noexcept {
    ~~~
}
```

Prefer Templates

- Since C++17, there is little reason to prefer a raw literal operator to a literal operator template.

```
// raw literal operator
constexpr Date
operator ""_Date(char const *cs) noexcept { ~~~ }

// literal operator template
template <char... cs>
constexpr Date
operator ""_Date() noexcept {
    constexpr array<char, sizeof...(cs)> d {cs...};
    ~~~
}
```

Google Again

- They're also upset because it's hard to use an ADT and not have to accept the UDLs that type defines.
- S(n)ide point: It's also hard to use an ADT and not have to accept the overloaded operators that type defines.
- If this is actually a problem, the best advice would seem to be to take it up with the designer of the type.
- Otherwise, put the UDLs for a type inside of a nested namespace, and force users to make them visible explicitly.

What's Wrong?

- Google: "Because they can't be namespace-qualified, uses of UDLs also require use of either using-directives (which we ban) or using-declarations (which we ban in header files <u>except when the imported names are part of the interface exposed by the header file in question</u>)."
- OK…

```
#include <string>
using std::string_literals::operator ""s;
~~~
std::string s = "xyz"s;
```

You Don’t Have to Use Using

- Note that a lot of our nifty coding techniques like the Making New Friends idiom won’t work with literal operators.
- Literal operators must be at namespace scope, and are found by ordinary lookup, not ADL.
- But there are other ways to introduce flexibility at namespace scope…

Possible Approach: Failed Overloading

- Suppose we'd like two versions of a literal, one that gives a compile-time error if used.

```
template <char... chars>      // version for _hash users
constexpr size_t operator "" _hash() noexcept {
    char const s[]{ chars... };
    return hash(arraytoi(s));
}

template <char... chars>      // version for _hash deniers
constexpr size_t operator "" _hash() noexcept { // error!
    static_assert(sizeof...(chars) == 0u,
                  "If you want to use _hash, say so!");
    return 0;
}
```

- As written, this is a redefinition.

Templates Add Flexibility

- User-directed SFINAE can disambiguate.

```
template <char... chars>     // version for _hash users
constexpr auto operator "" _hash() noexcept
    -> enable_if_t<hash_condition<chars...>, size_t>
    { ~~~ }

template <char... chars>     // version for _hash deniers
constexpr auto operator "" _hash() noexcept
    ->enable_if_t<!hash_condition<chars...>, size_t>
    { ~~~ }
```

- We have to provide a convenient mechanism to turn the `_hash` user-defined literal on and off.

Compile-Time Switch

- We can start with a condition that's always false.

```
template <bool b>
inline constexpr auto enable_hash = false;

template <char... chars>
constexpr auto hash_condition
    = enable_hash<sizeof...(chars) != 0>;
```

- A specialization, if present, turns the condition on.

```
template <>
inline constexpr auto enable_hash<true> = true;
```

A Macro!

- A macro can clean up the syntax (and wreak havoc on the semantics):

```
#ifndef enable_udl
    #define enable_udl( UD_SUFFIX ) \
        template <> inline constexpr auto \
        enable##UD_SUFFIX<true> = true
#endif
```

Flexibility

- A user can selectively turn on user-defined literals.

```
#include "hash.h"
#include "date.h"
enable_udl(_hash); // I want to use _hash, but not _Date
~~~
auto hval = 123456_hash;      // fine...
auto date = 2001'01'01_Date; // error!
```

Just a Suggestion

- This approach is clearly not perfect.
- For instance, it would probably e a good idea to include a namespace in the registration of a UDL:

```
enable_udl(std::string_literals, s);
```

- But going forward, it’s clear that some of the restrictions on literal operators, while not arbitrary are, well, restrictive.
  - They should be allowed to be templates, not for deduction but for the ability to apply templatesque coding techniques.
  - Literal operator templates should allow defaulted template parameters.
  - It would be useful to be able to specify (somehow!) a namespace qualification.

Opinions

- Used properly, user-defined literals can make code more readable and maintainable.
- Like overloaded operators, user-defined literals are typically part of an abstract data type's interface.
  - They should help to make the type "easy to use correctly and hard to use incorrectly."
  - ~~Do not u~~Use user-defined literals.
- Literal operator templates are often the user-defined literal implementation of choice for numeric or numeric-like literals because of their templaty flexibility.