EMBRACING USER DEFINED LITERALS *SAFELY*

for Types that Behave as though Built-in

Pablo Halpern phalpern@halpernwightsoftware.com

CppCon 2021

literal [lit-er-*uh*l]

adjective

1. in accordance with, involving, or being the primary or strict meaning of the word or words; not figurative or metaphorical: *the literal meaning of a word.*
2. following the words of the original very closely and exactly: *a literal translation of Goethe.*
3. true to fact; not exaggerated; actual or factual: *a literal description of conditions.*
4. being actually such, without exaggeration or inaccuracy: *the literal extermination of a city.*
5. (of persons) tending to construe words in the strict sense or in an unimaginative way; matter-of-fact; prosaic.

Source: dictionary.com

literal [lit-er-*uh*l]

noun

1. a typographical error, es~~p~~ecially involving a single letter.

Source: dictionary.com

literal (in C++)

noun

1. a single token in a program that represents a value of an integer, floating-point, character, string, Boolean, pointer, or user-defined type.

Examples of C++ literals

- Integer: `123, 456U, 0xfedcba10000LL, 0b1001UL`
- Floating point: `12.3, 4.56e-7F, 2.L`
- Character: `'a', L'b', u'c', U'd'`
- String: `"hello", L"goodbye", u8"see", u"you", U"later."`
- Boolean: `true, false`
- Pointer: `nullptr`
- User Defined: `59_min, 123'456'789'987'654'321'000_bigint, "[a-z]*"_regexp`

Contents

✅ What is a literal?

🔲 What are user-defined literals and why do we have them?

🔲 How do you define a new UDL suffix?

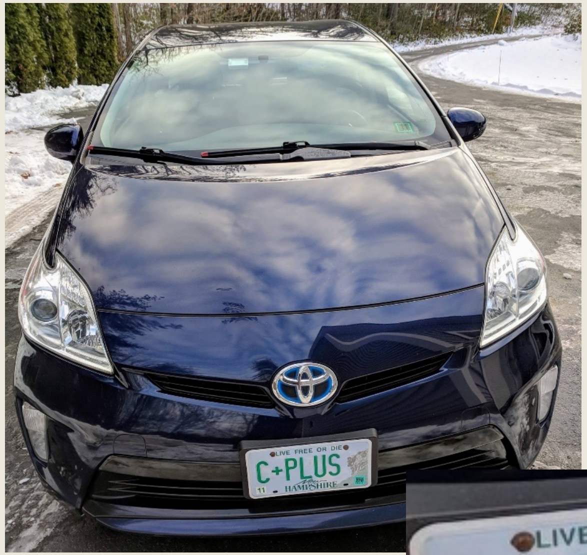
🔲 Use cases

🔲 Pitfalls

All examples are C++17 unless otherwise specified.

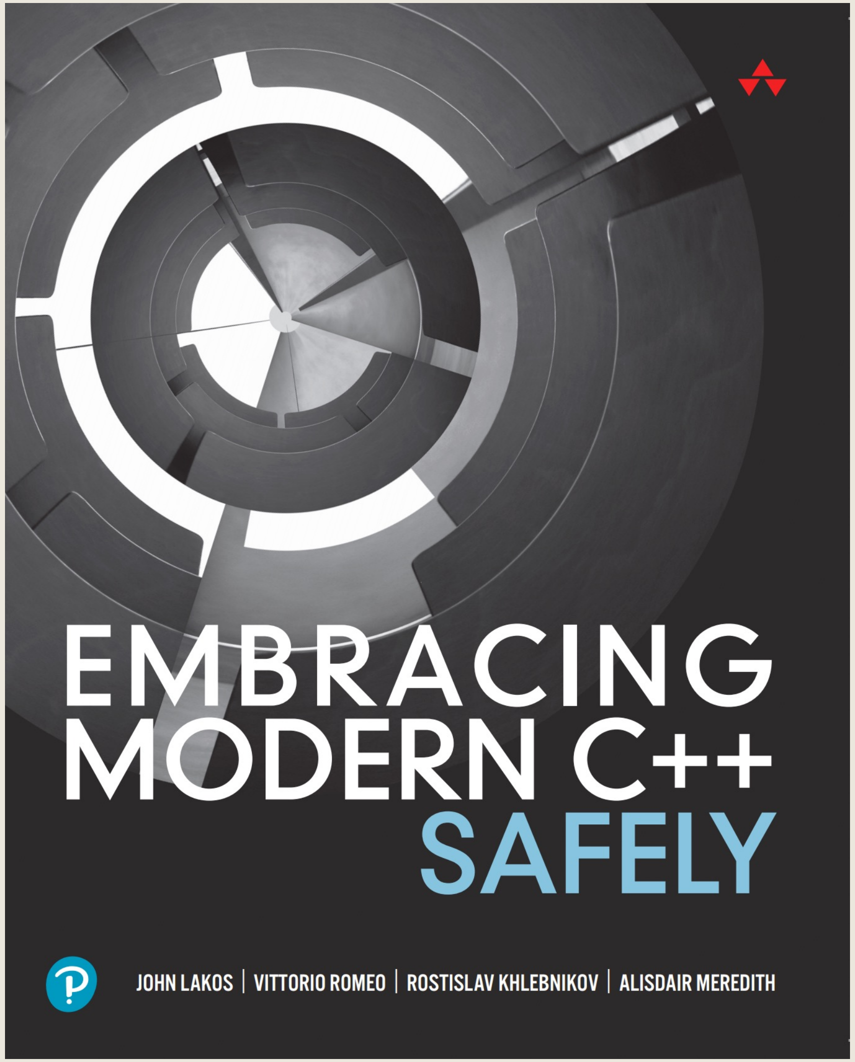
Interrupt me for ***clarifying*** questions only; please hold other questions until the end.

About me



- Independent software developer
- Member of the C++ Standards Committee
- Contributor to *Embracing Modern C++* ***Safely***
- Seventh year presenting at CppCon
- People brand me as a nerd despite my uber-sexy car (that, sadly, no longer has a C++ license plate)

Coming soon to a bookstore near you!



Talks in this series:

- Embracing User Defined Literals *Safely* for Types that Behave as though Built-in
  - *Pablo Halpern, Tuesday 9am*

You are here!

- Embracing (and also destroying) Variant Types *Safely*
  - *Andrei Alexandrescu, Thursday 9am*
- Embracing PODs *Safely* Until They Die
  - *Alisdair Meredith & Nina Ranns, Thursday 10:30am*
- Embracing \`noexcept\` Operators and Specifiers *Safely*
  - *John Lakos, Thursday 3:15pm*

WHAT ARE USER-DEFINED LITERALS AND WHY DO WE HAVE THEM?

What is a user-defined literal (UDL)?

- A user-defined literal is a literal having a program-defined meaning determined by a user-provided suffix.
- The *value* of a UDL can be a native type or a user-defined type.
- The definition (meaning) of the UDL suffix is provided by the user (or by the standard library).

Minimizing the divide between builtin and user-defined types

- Operator overloading allows the syntax for assigning, comparing, and streaming a `std::string` to be the same as for `int`:

```
extern unsigned    i, j; if (i < j) std::cout << i; // builtin type
extern std::string s, t; if (s < t) std::cout << s; // user-defined type
```

- If `5u` is a literal `unsigned` value, then why couldn't `"hello"s` be a literal `std::string` value?
- Now, it can!

Anatomy of a UDL

```
Temperature       room_temp =        20.0_C    ; // floating point UDL
IPv4Addr          loopback  = "127.0.0.1"_IPv4; // string UDL
std::chrono::hours half_day  =          12h     ; // integer UDL
const std::string greeting  =     "hello"s     ; // string UDL
```

"Naked" literal UDL Suffix

Suffixes without a leading underscore are reserved for the standard library.

Native literal suffixes (`U`, `L`, `LL`, `UL`, `ULL`, `LU`, `LLU`, and `F`) cannot be used as UDL suffixes for numeric UDLs.

Restrictions on UDLs

- The *naked literal* preceding the UDL suffix must be a syntactically-valid integer, floating-point, character, or string literal:
  - *OK:* `0x12_udl, 1.2e-2_udl, L'x'_udl, u8"yes"_udl`
  - *No:* `1.2.3_udl, nullptr_udl, false_udl`
- Numeric literals that do not fit in a native floating-point or integer type are OK (assuming an appropriate definition).
- String-token concatenation:
  - *Same suffix:* `"hello"_udl "world"_udl` → `"helloworld"_udl`
  - *Suffix + no suffix:* `"hello"_udl "world"` → `"helloworld"_udl`
  - *Mixed suffixes:* `"hello"_udlA "world"_udlB` → ERROR

The before days

- Constructors:

```
class IPv4Addr { ... constexpr explicit IPv4Addr(const char*); };
IPv4Addr loopback("127.0.0.1");
```

- Factory functions

```
class Temperature { ... };
constexpr Temperature celsius(double degreesC);   // factory function

auto room_temp = celsius(20.0);
```

- These approaches are still relevant in the days of UDLs!

So, aren't UDLs just syntactic sugar?

- You betcha!
- So is operator overloading.
- So are infix operators in general!
  - `a+b` is just syntactic sugar for `plus(a, b)`
- Syntactic sugar is about readability; it is not necessarily frivolous.
- BUT, UDLs can be used to obfuscate, just as operator overloading can!

DEFINING A UDL SUFFIX

The UDL operator

- `operator""_udl` defines a UDL suffix, `_udl` (whitespace discouraged).
- Template and argument lists depend on the form:
  - *Cooked UDL operator – The naked literal is evaluated at compile-time and passed into the operator as a value.*
  - *Raw UDL operator – The characters that make up the naked literal are passed to the operator as a* raw*, unevaluated string (numeric literals only).*
  - *Numeric UDL operator template – The characters that make up the naked literal are supplied,* as template arguments*, to the operator instantiation.*
  - *String UDL operator template – The naked literal is converted to a class type and supplied,* as a template argument*, to the operator instantiation.*

Namespaces for literals

- UDL operators are looked up using *ordinary name lookup*, no namespace qualifiers allowed.
- To avoid collisions between similar UDL suffixes, UDL operators are usually put in their own namespaces and imported with `using` directives:

```
namespace temperature_literals {
  constexpr Temperature operator""_deg(long double);
}
namespace geometry_literals {
  constexpr double operator""_deg(long double);
}

using namespace geometry_literals;
double right_angle = 90.0_deg;   // Unambiguously an angle, not a temperature
```

Cooked UDL Operators

Cooked UDL operators

- A.k.a. *Prepared-argument* UDL operators as used in *Embracing Modern C++ Safely*.
- The compiler fully evaluates the naked literal, then passes the resulting value to the UDL operator, which returns the resulting literal value; e.g.,

```
auto x = 1'234.5_udl; // equivalent to auto x = operator""_udl(1234.5L);
```

- Each Cooked UDL operator can have up to 12 overloads. Each overload can potentially return a different type.
  - *Integer and floating-point UDLs can each have their own overload*
  - *Character and string literals can each have up to 5 overloads, one for each built-in character width:* `char`, `wchar_t`, `char8_t`, `char16_t`, *and* `char32_t`.
- Integer overflow results in an error; floating-point over/underflow causes loss of precision.

Cooked UDL examples

```
struct Token {
  enum TokenType { internal, external };
  constexpr Token(unsigned val, TokenType type);
    // ...
};

constexpr Token operator""_token(unsigned long long v)
  { return Token(v, Token::internal); }

Token x = 1234_token;
auto  y = 0x2'0000'0000'0000'0000_token; // Error: ULL overflow
```

```
std::u8string operator""_u8str(const char*     s, std::size_t len);
std::u8string operator""_u8str(const char8_t*  s, std::size_t len);

auto hi    = "hi :-)"_u8str;            // calls first overload w/args "hi :-)", 6
auto smile = u8"Hi \U0001F600"_u8str;   // calls second overload w/args u8"Hi 😀", 7
```

Cooked UDL operator parameters

Example	UDL operator prototype
`123_udl`	`T operator""_udl(unsigned long long);`
`2.45e-6_udl`	`T operator""_udl(long double);`
`'a'_udl`	`T operator""_udl(char);`
`L'b'_udl`	`T operator""_udl(wchar_t);`
`u8'c'_udl`	`T operator""_udl(char8_t);` (since C++20)
`u'\u2190'_udl`	`T operator""_udl(char16_t);`
`U'\U00002190'_udl`	`T operator""_udl(char32_t);`
`"Hello"_udl`	`T operator""_udl(const char*, std::size_t);`
`L"World"_udl`	`T operator""_udl(const wchar_t*, std::size_t);`
`u8"Alpha\U0001F600"_udl`	`T operator""_udl(const char8_t*, std::size_t);`
`u"Beta \U0001F600"_udl`	`T operator""_udl(const char16_t*, std::size_t);`
`U"Gamma \U0001F600"_udl`	`T operator""_udl(const char32_t*, std::size_t);`

Cooked UDL: Nota bene

- Only the 12 signatures specified are allowed. Other integer and floating-point types cannot be used as parameters:

```
int operator""_udlA(int);        // Error: int is not a valid UDL parameter type
int operator""_udlB(double);     // Error: double is not a valid UDL parameter type
```

- When interpreting a UDL, promotions and conversions are *not* applied; arguments must match parameters *exactly*:

```
int operator""_udlC(long double);   // Floating-point UDL operator
int operator""_udlD(wchar_t);       // Wide-character UDL operator

auto c = 123_udlC;  // Error: can't find operator""_udlC(unsigned long long)
auto d = 'd'_udlD;  // Error: can't find operator""_udlD(char)
```

Raw UDL Operators

Raw UDL operators

- For numeric UDLs only
- The prototype for a raw UDL operator for UDL suffix _*udl* is

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

- The compiler syntactically validates *but does not evaluate* the naked literal; it passes the *raw* characters as a null-terminated string to the UDL operator.

```
auto x = 1'234.5_udl; // equivalent to auto x = operator""_udl("1'234.5");
```

- The UDL operator can parse the naked literal anyway it wants.

Raw UDL operator example: Base 3 int

```
constexpr int operator"" _3(const char* digits) {
  int result = 0;

  while (*digits) {
    result *= 3;
    result += *digits - '0';
    ++digits;
  }

  return result;
}

static_assert(21_3 == 7, "");

int i1 = 22_3;                       // OK, returns `(int) 8`
int i2 = 23_3;                       // Bug, returns `(int) 9`
int i3 = 21.1_3;                     // Bug, returns `(int) 58`
int i4 = 2221110002221110022_3;  // Bug, too big for 32-bit `int`
```

Base 3 int with error detection

```
constexpr          int operator"" _3(const char *digits)
{
  int ret = 0;
  for (char c = *digits; c; c = *++digits) {
    if ('\'' == c) continue;  // Ignore digit separator.
    if (c < '0' || '2' < c)
      throw std::out_of_range("Invalid base-3 digit");
    if (ret >= (std::numeric_limits<int>::max() - (c - '0')) / 3)
      throw std::overflow_error("Integer overflow");
    ret = 3 * ret + (c - '0');  // Consume `c`
  }
  return ret;
}

int           i1 = 1'200_3;      // OK, returns 45
constexpr int i2 = 23_3;         // Error detected at   compile time
int           i3 = 21.1_3;       // Error detected at run     time
int i4 = 22211100022211100022_3; // Error detected at run     time
```

Force compile-time error detection

C++20 feature

```
constexprconsteval int operator"" _3(const char *digits)
{
  int ret = 0;
  for (char c = *digits; c; c = *++digits) {
    if ('\'' == c) continue;  // Ignore digit separator.
    if (c < '0' || '2' < c)
      throw std::out_of_range("Invalid base-3 digit");
    if (ret >= (std::numeric_limits<int>::max() - (c - '0')) / 3)
      throw std::overflow_error("Integer overflow");
    ret = 3 * ret + (c - '0');  // Consume `c`
  }
  return ret;
}

int           i1 = 1'200_3;       // OK, returns 45
constexpr int i2 = 23_3;          // Error detected at    compile time
int           i3 = 21.1_3;        // Error detected at runcompile time
int i4 = 2221110002221100022_3;   // Error detected at runcompile time
```

Raw UDL operators: Nota Bene

- Integer and floating-point overflow/underflow do not occur prior to calling the UDL operator. Raw UDL operators are thus suited for extended-precision numeric types.
- There is only one raw UDL operator signature for a given UDL suffix; it cannot be overloaded separately for integer vs. floating point types.
- If a matching cooked UDL operator is found, it is preferred over the raw one.

```
constexpr int operator""_udl(long double) { ... } // (1) Cooked UDL operator
constexpr int operator""_udl(const char*) { ... } // (2) Raw UDL operator

int x = 12._udl;   // Evaluates overload (1)
int y = 123_udl;   // Evaluates overload (2)
```

Numeric UDL operator templates

Numeric UDL operator templates

- For numeric UDLs only
- The prototype for a numeric UDL operator template for UDL suffix _*udl* is

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

- The compiler syntactically validates *but does not evaluate* the naked literal; it *instantiates* the template with the sequence of characters in the naked literal:

```
auto x = 1'234.5_udl;
  // equivalent to auto x = operator""_udl<'1', '\'', '2', '3', '4', '.', '5'>();
```

- The return type can be fixed or determined from the naked literal using template metaprogramming.

Base 3 int revisited (helper templates)

```
constexpr long long llmax = std::numeric_limits<long long>::max();

template <long long partial>
constexpr long long base3i() { return partial; /* base case */ }

template <long long partial, char c0, char... c>
constexpr long long base3i() {  // recursively compute base-3 integer
  if constexpr ('\'' == c0)
    return base3i<partial, c...>();
  else {
    static_assert('0' <= c0 && c0 < '3', "Invalid based-3 digit");
    static_assert(partial <= (llmax - (c0-'0')) / 3,
                  "`long long` overflow");
    return base3i<3 * partial + c0 - '0', c...>();
  }
}
```

Integer UDL operator template example: base 3 integer literals

- Using the helper templates, the UDL operator template would be simple:

```
template <char... c>
constexpr long long operator "" _3() { return base3i<0, c...>(); }
```

- But we can do one better, returning `int` in most cases, but `long long` if the result is too big to fit in an `int`:

```
constexpr int imax = std::numeric_limits<int>::max();

template <char... c>
constexpr auto operator "" _3()
  -> std::conditional_t<base3i<0, c...>() <= imax, int, long long>
{
  return base3i<0, c...>();
}
```

UDL operator templates: Nota Bene

- As in the case of raw UDL operatators, integer and floating-point overflow/underflow do not occur prior to instantiating the UDL operator template.
- There is only one numeric UDL operator template signature for a given UDL suffix; it cannot be overloaded separately for integer vs. floating point types.
- If a matching cooked UDL operator is found, it is preferred over the template. If a matching raw UDL operator is found, it is ambiguous:

```
constexpr int operator""_udl(long double) { ... } // (1) Cooked UDL operator
constexpr int operator""_udl(const char*) { ... } // (2) Raw UDL operator
template <char...>
constexpr int operator""_udl() { ... }            // (3) UDL operator template

int x = 12._udl;   // Evaluates overload (1)
int y = 123_udl;   // Ambiguous overload (2) or (3)
```

Comparing template UDL operators to raw UDL operators

- Return type can be determined based on input.
- Can force evaluation (and error detection) at compile time without C++20 `consteval`.
- Often requires more complex implementation – typically involving template metaprogramming – than either cooked or raw UDL operators.

String UDL operator templates

String UDL operator templates

- C++20 Only
- The prototype for a string UDL operator template for UDL suffix *_udl* is

```
template <StructType S> T operator""_udl();
```

- The type, `StructType`, must be a *structural class type* – a `struct` having:
  - *a (defaulted or user-provided)* `constexpr` *constructor*
  - *a (defaulted or user-provided)* `constexpr` *destructor*
  - *members and base classes that are all* `public` *and structural*
- `StructType` must be implicitly convertible from a native string literal.
- If `StructType` is a class template, its template arguments must be deducible when initialized from a native string literal using CTAD.

Example of a usable structural type

```
template <typename CharT, std::size_t N>
struct StrLiteralProxy
{
  constexpr StrLiteralProxy(const CharT (&s)[N])
      { std::copy(std::begin(s), std::end(s), std::begin(m_data)); }

  constexpr std::size_t size() const { return N - 1; }
  constexpr const CharT* data() const { return m_data; }

  CharT m_data[N];
};

StrLiteralProxy x = "hello"; // Deduced as StrLiteralProxy<char, 6>
StrLiteralProxy y = u"yes";  // Deduced as StrLiteralProxy<char16_t, 4>
```

Cannot store pointer to s!

Example of a string UDL operator template: IP addresses

```
struct IPV4Addr
{
  static constexpr bool isIpV4Format(const char* str);
  explicit constexpr IPV4Addr(const char* str) { }
  ...
};
struct IPV6Addr
{
  explicit constexpr IPV6Addr(const char* str) { }
  ...
};

template <StrLiteralProxy S> constexpr auto operator""_IP() {
  return std::conditional_t<IPV4Addr::isIpV4Format(S.data()),
                              IPV4Addr, IPV6Addr>(S.data());
}
```

IP addresses (continued)

```
auto v4 = "1.2.3.4"_IP;   // IPv4Addr
auto v6 = "1:2::3:4"_IP; // IPv6Addr
```

- The main benefit of a string UDL operator template over a cooked string UDL operator is the ability to perform template metaprogramming on the value, e.g., to select a type at compile time.
- If a matching cooked UDL operator is found, the *operator template* is preferred:

```
//  (1) Cooked UDL operator
constexpr int operator""_udl(const char*, std::size_t) { ... }
//  (2) String UDL operator template (effectively hides overload (1))
template <StrLiteralProxy> constexpr int operator""_udl() { ... }

constexpr int h = "hello"_udl;   // Evaluates overload (2)
constexpr int b = u8"bye"_udl;   // Evaluates overload (2)
```

STANDARD LIBRARY UDLS

About UDLs in the standard library

- User-defined literals were added to the language in C++11, but were not used in the standard library until C++14.
- All literals in the standard library are in an inline sub-namespace of the inline namespace, `std::literals`; they must be imported into the current scope by means of a `using` directive:
  - `using namespace std;` *imports everything in the standard library, including all the literals.*
  - `using namespace std::literals;` *imports all the standard literals.*
  - `using namespace std::literals::`*sub-namespace* *imports just the literals in the specified sub-namespace.*

```
using namespace std::literals::string_literals;
auto s = "hello"s; // std::literals::string_literals::operator""s is in scope
```

String UDLs

- In sub-namespace **`string_literals`**, UDL suffix **`s`** yields a **`basic_string`**:

```
using namespace std::literals::string_literals;
auto s1 = "hello"s;  // std::string
auto s2 = u8"bye"s;  // std::u8string (basic_string<char8_t>)
```

- In sub-namespace **`string_view_literals`**, UDL suffix **`sv`** yields a **`basic_string_view`**:

```
using namespace std::literals::string_view_literals;
auto s1 = "hello"sv;  // std::string
auto s2 = u"bye"sv;   // std::u16string_view (basic_string_view<char16_t>)
```

Imaginary number UDLs

- In sub-namespace complex_literals:

UDL Suffix	Result type
`i`	`complex<double>`
`if`	`complex<float>`
`il`	`complex<long double>`

- Both integer and floating-point literals accepted; integers are converted to floating point.
- The resulting value has a zero real part and the specified imaginary part:

```
using namespace std::literals::complex_literals;
auto x = 4if;           // complex<float>(0, 4.0)
auto y = 5.0 - 3.2i;   // complex<double>(5.0, -3.2)
```

Chrono duration UDLs

- In sub-namespace **chrono_literals**:

UDL suffix	Integer literal result type	floating-point literal type
h	**chrono::hours**	Appropriate floating-point instantiation of **chrono::duration**
min	**chrono::minutes**	
s	**chrono::seconds**	
ms	**chrono::milliseconds**	
us	**chrono::microseconds**	
ns	**chrono::nanoseconds**	

- Namespace `std::`**`chrono`**`::chrono_literals` is an alias for `std::`**`literals`**`::chrono_literals`.

```
using namespace std::chrono::chrono_literals;
constexpr auto elapsed = 8min + 5.2s;
```

Chrono day and year UDLs
(since C++20)

- Also in sub-namespace `chrono_literals`:

UDL suffix	Result type
`d`	`chrono::day`
`y`	`chrono::year`

- Both are integer literals.
- Constants are defined for the days of the week and months of the year, but they do not have a numeric literal representation.

USE CASES

The problem: type sinks

```
typedef int part_number;

extern void add_to_inventory(part_number pn, int quantity);

add_to_inventory(90042, 2);   // OK. Add 2 units of part 90042
add_to_inventory(2, 90042);   // Oops! Add 90042 units of part
add_to_inventory(01773, 1);   // Oops! Add 1 unit of part 1019, not part 01173
```

- `int` is a *type sink*: as a function parameter, it can match many semantically unrelated values.

Strong typedefs: the solution

```
enum part_number : int { };   // strong typedef

namespace part_literals {
part_number operator""_part(const char* n) {
  return part_number(std::strtol(n, nullptr, 10));
}
}

extern void add_to_inventory(part_number pn, int quantity);

using namespace part_literals;
add_to_inventory(90042_part, 2);   // OK. Add 2 units of part 90042
add_to_inventory(2, 90042_part);   // Argument mismatch error! Won't compile.
add_to_inventory(01773_part, 1);   // OK! Add 1 unit of part 01173
```

- UDLs can make strong typedefs more natural and less error prone to use.

Extended numeric types: Arbitrary-precision integers

```
class BigInt { ... };

namespace bigint_literals {
BigInt operator""_bigint(const char* digits); // raw UDL operator
}

using namespace bigint_literals;
BigInt b = 184'467'440'737'095'516'150_bigint;
```

Extended numeric types: Decimal fixed-point numbers

```
template <int precision> class FixedPoint {
  long long d_data;  // 64-bit value = d_data / pow(10, precision)
public:
  static constexpr FixedPoint makeRaw(long long data);
  ...
};

template <long long rawVal, int precision, char... c>
struct MakeFixedPoint;  // Metafunction to compute fixed-point number from character list

namespace fixedpoint_literals {
template <char... c> auto operator""_fixed() { // UDL template
  return
    MakeFixedPoint<0,std::numeric_limits<int>::min(),c...>::makeValue();
}
}
```

Extended numeric types: Decimal fixed-point numbers (continued)

```
template <long long rawVal, int precision>
struct MakeFixedPoint<rawVal, precision> {   // Base case; no more characters
  static constexpr auto makeValue() {
    return FixedPoint<(precision < 0) ? 0 : precision>::makeRaw(rawVal);
  }
};

template <long long rawVal, int precision, char... c>
struct MakeFixedPoint<rawVal, precision, '.', c...> // match decimal point
    : MakeFixedPoint<rawVal, 0, c...> {
  static_assert(precision < 0);
};

template <long long rawVal, int precision, char... c>
struct MakeFixedPoint<rawVal, precision, '\'', c...> // match digit separator
    : MakeFixedPoint<rawVal, precision, c...> { };
```

Extended numeric types: Decimal fixed-point numbers (continued)

```
template <long long rawVal, int precision, char c0, char... c>
struct MakeFixedPoint<rawVal, precision, c0, c...>
    : MakeFixedPoint<rawVal * 10 + (c0 - '0'), precision + 1, c...>
{
  static_assert('0' <= c && c <= '9');
  static_assert(std::numeric_limits<long long>::max() - (c0 - '0')) / 10
                >= rawVal, "Fixed-point overflow");  // Overflow check
};

using namespace fixedpoint_literals;
auto x = 1_fixed;      // FixedPoint<0>
auto y = 1.2_fixed;    // FixedPoint<1>
auto z = 1.234_fixed;  // FixedPoint<3>
auto e - 1.2e5_fixed;  // Error: invalid character 'e'
```

Special-format string-like classes

```
constexpr IPv4Addr operator""_IPv4(const char*, std::size_t);

auto loopback = "127.0.0.1"_IPv4;   // Verified IPv4 address
auto other    = "127.300.0.0"_IPv4; // Error: invalid IPv4 address


constexpr UUIDv4 operator""_UUID(const char*, std::size_t);

UUIDv4 Fred = "eeec1114-8078-49c5-93ca-fea6fbd6a280"_UUID;  // OK
UUIDv4 Bad  = "123x"_UUID;   // Error: bad UUID format
```

Physical units

```
namespace si_literals
{
  constexpr Distance operator""_m  (long double meters);
  constexpr Distance operator""_cm (long double centimeters);
  constexpr Time     operator""_s  (long double seconds);
  constexpr Mass     operator""_g  (long double grams);
  constexpr Mass     operator""_kg (long double kg);

  constexpr Speed    operator""_mps(long double mps);
  constexpr Energy   operator""_j  (long double joules);
}
```

Physical units (continued)

```
using namespace si_literals;
auto d3   = 15.0_m;    // distance in meters
auto t3   = 4.0_s;     // time in seconds
auto s3   = d3 / t3;   // speed in m/s (meters/second)
auto m3   = 2045.0_g;  // mass expressed as g but stored as Kg
auto m3Kg = 2.045_kg;  // mass expressed as kg
```

Mateusz Pusz explores the topic of a comprehensive physical units library in his CppCon 2020 talk, ***A Physical Units Library For the Next C++*** (https://youtu.be/7dExYGSOJzo).

PITFALLS!

Does your code do what you think it does?

- Raw UDL operators and UDL operator templates must parse their inputs. Parsing errors lead to program errors:

```
short operator""_short(const char *digits) // Returns a short
{
  short result = 0;
  for (; *digits; ++digits)
    result = result * 10 + *digits - '0';

  return result;
}

short s1 = 123_short;   // OK, value 123
short s2 = 123._short;  // Bug, `.` treated as digit value -2
short s3 = 1'234_short; // Bug, `'` treated as digit value -9
```

Can you obfuscate that better?

- Raw UDL operators can parse numeric input in a way that is completely unrelated to normal numbers, but is **`192'168'0'1_IPv4`** really easier to read than **`"192.168.0.1"_IPv4`**?
  - *The first is a numeric UDL that interprets the single quote as an octet separator, contrary to its normal use as a digit separator.*
  - *The second is a string UDL that does not conflict with conventional interpretations.*
- String UDLs can potentially be interpreted as a list of items, but is **`"0,f0,10"_rgb`** superior to **`rgb(0, 0xf0, 0x10)`**?
  - *The first requires a string UDL operator to parse multiple numeric subparts.*
  - *The second is a simple constructor call.*
  - *Which is less error prone?*

A lot of work to support "magic" values

- We're told that hard-coded values in a program should be given names:

```
verticalOffset(std::sin(0.241));       // Bad: uses magic number 0.241
constexpr double mastAngle = 0.241;    // Define a named constant
verticalOffset(std::sin(mastAngle));   // Preferred: uses named constant
```

- Shouldn't the same be true of UDTs?

```
setGoalSpeed(686_mps);        // Magic value, 686
constexpr Speed mach1{343};   // 343 mps
setGoalSpeed(2 * mach1);      // Preferred style: no magic number or UDL
```

- The most common magic value represents some notion of *zero* or *empty*. What is the point of defining a UDL that will be used for only a single value?

```
Thing operator""_thing(unsigned long long);   // UDL for thing
Thing a = 0_thing;                            // Uses magic number 0
constexpr Thing nullThing{0};   // Define a named constant
Thing b = nullThing;            // Arguably clearer than 0_thing; no UDL
```

Wait for the sign

- Both built-in and user-defined numeric literals are always non-negative.

```
int  x = -5;   // Evaluated as operator-(5)
using namespace std::literals::chrono_literals;
auto t = -5m;  // Evaluated as operator-(std::chrono::minutes{5})
```

- Beware conversions that should not be negated!

```
// Normalize all temperatures to Kelvin
constexpr double celsius(double c) { return c + 273.15; }
constexpr double fahrenheit(double f) { return (f-32)*5/9 + 273.15; }
constexpr double operator""_C(long double c) { return celsius(c); }
constexpr double operator""_F(long double f) { return fahrenheit(f); }

double t1 = celsius(-10);   // OK, returns 263.15 degrees Kelvin
double t2 = -10_C;          // Oops! Returns 283.15 degrees below absolute zero!
```

Aside: points vs. deltas

- Many measurements distinguish between absolute points (or positions) and deltas from one point to another.
  - *Point + Point => ERROR*
  - *Point - Point => Delta*
  - *Delta +/- Delta => Delta*
  - *Point +/- Delta => Point*
- A single unit will often have both possible meanings; e.g. -10℃ could be a temperature point (10° below freezing) or a delta (change relative to some temperature point).
  - *The potential confusion is most problematic when the origin (0) of the point unit is arbitrary.*

Avoiding point/delta confusion

- Create separate types for point and delta quantities
  - *E.g.,* **`std::chrono`** *has separate types for* **`time_point`** *(point) and* **`duration`** *(delta).*
- Typically, only deltas will benefit from UDLs. When defining UDLs for points, make that clear from the suffix
  - *E.g.,* **`_mile_marker`** *(point UDL) and* **`_mile`** *(delta UDL)*
  - *E.g.,* **`_tempC`** *(point UDL) and* **`_C`** *(delta UDL)*

CONCLUSIONS

Conclusions

- UDLs were added to C++11 to close a gap between the syntax used to manage values of built-in types vs. user-defined types.
  - *The original use-case, decimal floating-point values, is still in committee.*
- There are four different formats for UDL operators; each more powerful but more difficult to define than the one before.
- The UDL system is extremely powerful and flexible.
- Just as in the case of operator overloading, it can be abused.

QUESTIONS?