User-defined Literals
(aka. Extensible Literals (revision 4))

Ian McIntosh, Michael Wong, Raymond Mak, Robert Klarer, Jens Maurer, Alisdair Meredith, Bjarne Stroustrup, David Vandevoorde

ianm@ca.ibm.com
michaelw@ca.ibm.com
rmak@ca.ibm.com
rklarer@ca.ibm.com
jens.maurer@gmx.net
public@alisdairm.net
bs@cs.tamu.edu
daveed@edg.com

Document number: N2750-08-0260
Date: 2008-08-22
Project: Programming Language C++, Core Working Group
Reply-to: Michael Wong (michaelw@ca.ibm.com)
Revision: 4

Abstract

This paper is Revision 4 of n1892 [n1892], n2282 [n2282], n2378 [n2378] and proposes additional forms of literals using modified syntax and semantics to provide user-defined literals. User-defined literals allow user-defined classes to provide new literal syntax, a feature previously available only for built-in types. It increases compatibility with C99 and future C enhancements, as well as more flexible C++ literals.

The existing set of (C++03) literals is extended: Any literal may contain a user-defined suffix. Examples include **`"Hi"s`**, **`1.2i`**, and **`23_units`**. A user-defined "literal operator" function defines the mapping of such user-defined literals to actual values. User-defined literals can produce values of both built-in types (e.g., **`double`**) and user-defined types (e.g., class types).

The proposal requires fairly localized changes to the core language.

1 History

User-defined literals (previously called “extensible literals”) have been presented repeatedly since the 2005 Mont Tremblant meeting and the basic idea was always well received. However, it was not until the Oxford meeting that we found a way to address all implementation and specification difficulties. In Oxford (2007), David Vandevoorde offered a solution which is essentially the one presented here.

This solution was presented in Toronto (2007) and was accepted by EWG with the provision that wording be added. This paper presents that wording, a summary of all the constraints discussed in Toronto, and a description of the basic idea for people who may have gotten lost in the details and the revision history.

Revision history:

Revision 3:

- Added wording, and updated with constraints discussed in EWG in Toronto, July 2007. Changed the term “extensible literal” to “user-defined literal”.

Revision 2:

- Update on a Syntax suggested by Daveed Vandevoorde and Dave Abrahams from the April, 2007 Oxford meeting

2 The Problem

C++ provides literals for its basic data types: integer literals, floating-point literals, character literals, string literals, and Boolean literals[1]. The type of the value of such a literal is primarily determined from its syntactic form (e.g., the presence of a decimal point, exponent, or alphabetic suffix). (More rarely, the magnitude may modify the type implied by the syntactic form.) The type and implementation determine the data representation.

To add a new data type to a non-extensible language such as C [C99], the language syntax and semantics must be modified by adding the type's name, the type's operations, and where appropriate the type's literal syntax (e.g., the new suffix **`dd`** for decimal floating point literals).

To add a new data type to an extensible language such as C++ [C++03], the preferred approach is to leave the language unchanged and define a new class implementing the type's operations. Until now, however, there has been no way to introduce new literal forms (and hence no way to achieve complete source compatibility with C using library-based approaches). This paper supports two basic principles of C++ design:

---

[1] Boolean literals are, however, different because they have a small set of values that are handled using keywords. They are ignored in what follows.

- User-defined types should have all the same support facilities as built-in types, and [12].
- C compatibility should be maintained as far as possible [9,10,11,16].

The existing mechanisms work well when existing literals (integers, floating-point and string literals) are suitable. Other proposals [1,2] extend that to classes which are aggregates of existing types by adding user-defined literals formed by grouping basic data type literals; e.g., **complex(1,2)**. This proposal allows additional forms of non-standard literals and uses an operator-based mechanism to provide extensible user-defined literals.

2.1 Goals

A major goal for user-defined literal suffixes is to handle every suffix currently in C or proposed for C (without changing the C++ core language). A second goal is to handle every suffix in common extensions to C++. Examples are:

```
123.4567890123df      // decimal floating-point
"hello!"s             // std::string (rather then C-style string)
3.4i                  // imaginary type (for complex)
```

Our design does *not* provide a mechanism for specifying prefixes for literals – doing that appears to require major surgery on the lexical rules, whereas this proposal aims at working well with existing implementations.

The goal for data representation is to be able to produce data for every existing, proposed or future numeric or string data format, including integer, binary floating-point and decimal floating-point, in any reasonable size or precision and representation. For example:

```
101011100011b   // binary literals
123km           // unit is kilometers
1782493728347927857293847198237182379123981123791 2x
                // numbers with arbitrary range/precision
```

Important use cases include literals where the meaning of characters differ from the standard, such as binary and decimal floating point above, and literals representing values larger than any built-in type.

3 The Proposed Solution

In this section we informally but completely present the newly-proposed language feature.

3.1 Operators for new lexical forms

We propose that any built-in literal form not already containing a suffix (e.g., `23` but not `23LL`) concatenated with an identifier (which forms the **suffix**) be an acceptable literal form. Examples include:

```
0p
2_pi
99999LLX
0xAE3_ROM
1.2e-3DF
"Hello!"s
L"I18N"ws
U'\u2233'_glyph32.
```

Any such literal form that is not a built-in literal form is called a **user-defined literal**. Note that there is no *a priori* length limit to user-defined literals: This allows for literals that map to an arbitrary-precision number class.

User-defined string literals can be concatenated like other literals, but the string resulting from a concatenation cannot involve component strings that have different suffixes. On the other hand, to make certain macros more useful, we allow mixing built-in string literals with user-defined string literals. For example:

```
L"log: " "I/O error"s   // Same as L"log: I/O error"s
"log: "x "I/O error"s   // Error: different suffixes
```

Furthermore, we propose that every user-defined literal be interpreted as an unqualified call to a new kind of operator—called **literal operator**—that has the following general form:

```
X operator "suffix" ( <parameters> );
```

where **X** is an arbitrary return type, ***suffix*** is an identifier corresponding to the suffixes of the literals that map to a call to this operator, and ***<parameters>*** is a parameter list to be described below. For example, the list of sample literals above would map onto calls to literal operators as follows:

```
0p                  calls operator"p"(...)
2_pi                calls operator"_pi"(...)
99999LLX            calls operator"LLX"(...)
0xAE3_ROM           calls operator"_ROM"(...)
1.2e-3DF            calls operator"DF"(...)
"Hello!"s           calls operator"s"(...)
L"I18N"ws           calls operator"ws"(...)
U'\u2233'_glyph32   calls operator"_glyph32"(...)
```

The arguments to the call are omitted at this point, but they are of course a function of what precedes the prefix. They will be determined in what follows.

Except for their name, literal operators are ordinary namespace-scope functions. They can be declared with **inline** or **constexpr**, have internal or external linkage, or have their address taken. The **constexpr** option in particular means that some user-defined literals may be ROMable or may participate in static (as opposed to dynamic) initialization.

3.2 Raw vs. Cooked Literals

We call **raw literal** the sequence of characters that form a literal. This sequence of characters often directly appears in the source, but it can also result from the transformations required by the first six phases of translation (which include macro expansions and string literal concatenations).

For built-in literals, the **cooked form** is the typed value that the literal represents. For example, the literal **`12`** corresponds to a cooked form that is an **`int`** of value twelve. For user-defined string literals and character literals, the cooked form is the cooked form of the literal obtained by leaving out the suffix. For example:

`"Hello!"s` cooks to **`"Hello!"`**, which is an array
`{ 'H', 'e', 'l', 'l', 'o', '!', '\0' }`

For user-defined numerical literals that are suffixed integer literals, the cooked value is the value of the literal obtained by replacing the suffix by ULL[2]. For example:

`0p` cooks to the value and type of **`0ULL`**

Similarly, user-defined numerical literals that are suffixed floating literals, the cooked value is the value of the literal obtained by replacing the suffix by **`L`** (thus obtaining a value of type **`long double`**). For example:

`1.2e+3DF` cooks to the value and type of **`1.2e+3L`**

For many applications it is sufficient (and convenient!) to transform the cooked value and we don't really care what its raw form was. For example, we may like to map

`"Hello!"s` to **`std::string("Hello!")`**

and

`2_pi` to **`2*pi`**

Sometimes, however, significant information is lost in the cooked form. For example, the cooked form of **`1.2e-3DF`** may not exactly equal the mathematical value 0.0012, whereas a decimal floating-point type to which this literal is intended to map would in fact represent that value exactly. Similarly, a very long integer literal

`3523175843937492387423498723498720984237832247x`

can be useful when mapped to an arbitrary-precision class type, but might not have a valid cooked value. To deal with such numerical cases, we want the ability to transform the raw literal form of those literals.

To address these observations, we therefore propose one form of literal operator that transforms the raw form of numerical (i.e., integer and floating-point) literals, and a variety of literal operator forms that deal with the different types of cooked literal types.

---

[2] I.e., a value of type **`unsigned long long`**. Note that integer literals cannot be negative (the – sign is not part of the literal, but a separate token).

3.3 The raw-form operator

The raw-form literal operator has the following form:

```
X operator "suffix" (char const*);
```

For example, if

```
unsigned long long operator"B"(char const*);
```

is the only **operator"B"** in scope, then the literal

```
01100001000B
```

is treated as a call

```
operator"B"("01100001000")
```

i.e., the raw form of the token (minus its suffix) is passed as a null-terminated string to the (raw) literal operator, which can then parse the characters of that token into a meaningful value. The literal

```
1.2B
```

results in a call

```
operator"B"("1.2")
```

which might e.g. throw an exception because the raw form does not correspond to a sequence of 0s and 1s.

On the other hand, the literal

```
"1001"B
```

results in a compilation error, because string literals never map onto calls to a raw-form literal operator. This restriction is added to avoid specification and implementation difficulties that arise from the details of the phases of translation. For example, the literal

```
"Hello, " L"Worl\u0044!"
```

may be indistinguishable from

```
L"Hello, World!"
```

at the end of phase 6. The latter could easily be made the basis for the raw form, but we expect that that would be surprising to programmers. Making the former the raw form would require significant surgery in the phases of translation, and might very significantly increase the implementation cost for some compiler vendors. Since we are not aware of compelling use cases, we avoid the issue altogether by not defining mapping from string literals and character literals to the raw form literal operators.

Finally, it may be worth pointing out that:

```
0x1B
```

is a built-in hexadecimal literal; not the literal **0x1** with a suffix **B**.

In addition to the ordinary form described above, we also propose a variadic template form:

```
template<char...> X operator "suffix" ();
```

For example, if only

```
template <char...> unsigned long long operator"B"();
```

is in scope, then

```
00101B
```

results in a call

```
operator"B"<'0', '0', '1', '0', '1'>()
```

(Note the lack of a terminating null character in that case.)

The variadic template form combined with **constexpr** allows for the use of template metaprogramming to produce compile-time values from the raw form.

For a given suffix, declaring both the ordinary and variadic template form in the same scope is an error. Furthermore, if lookup during literal processing finds two applicable raw forms, an ambiguity error occurs. For example:

```
namespace N1 {
  template <char...> A operator"A"();
}

namespace N2 {
  A operator"A"(char const*);
}

using namespace N1;
using namespace N2;

A a = 100A;  // Error: Ambiguous!
```

3.4 The cooked-form operators

The cooked-form literal operator for user-defined integer literal has the following form:

```
X operator "suffix" (unsigned long long);
```

For example, if

```
constexpr
long double operator"_pi"(unsigned long long m) {
  return m*pi;
}
```

is in scope, then

```
2_pi
```

results in a call

```
operator"_pi"(2ULL)
```

Cooked-form operators take precedence over the corresponding raw-form operator. So if the following two declarations were in scope:

```
long double operator"_pi"(unsigned long long);
long double operator"_pi"(char const*);
```

then

```
2_pi
```

would still result in a call

```
operator"_pi"(2ULL)
```

The cooked-form literal operator for a user-defined floating literal has the form:

```
X operator "suffix" (long double);
```

For user-defined string literals, the following cooked-form operator forms are invoked:

```
X operator "suffix" (char const*, size_t);
X operator "suffix" (wchar_t const*, size_t);
X operator "suffix" (char16_t const*, size_t);
X operator "suffix" (char32_t const*, size_t);
```

when respectively the string is narrow, has an **L** prefix, a **u** prefix, or a **U** prefix. The number of characters in the cooked literal (not including the terminating null) is passed as the second argument[3].

For example, assuming

```
wstring operator"ws"(wchar_t const *str, size_t n) {
  return wstring(str, n);
}
```

is in scope

```
L"I18N"ws
```

results in a call

```
operator"ws"(L"I18N", 4)
```

---

[3] The length is passed for the cooked-form literal operator because it is expected that it can generally use that value directly to e.g. allocate memory for the string. The raw-form literal operator, in contrast is much less likely to be able to use the extra argument that way. This difference conveniently makes the two cases distinguishable.

Similarly (assuming raw string literals as proposed in N2295),

```
LR"--[The name is "C++".]--"ws
```

results in a call

```
operator"ws"(L"The name is \"C++\".", 18)
```

However,

```
"I18N"ws
```

results in an error, because only a wide-string (prefix **L**) operator is in scope.

User-defined character literals are handled similarly using different cooked-form literal operators:

```
X operator "suffix" (char);
X operator "suffix" (wchar_t);
X operator "suffix" (char16_t);
X operator "suffix" (char32_t);
```

3.5 An idiom

User-defined literal operators are looked up like any other function. However, if called through literal forms they are called using built-in types and are therefore never found through argument-dependent lookup.

It is tempting therefore to declare user-defined literal operators in the global namespace, or to place a global using-declaration in the global namespace. Unfortunately, that is a recipe for declaration conflicts even for code that might not use the literal forms.

An alternative idiom consists in declaring literal operators in their own namespace, and adding a "using directive" in the global namespace to make them visible to client code. Conflicts are delayed until the point of use in this way.

For example:

```
// File a.h:
namespace A {
  class X { … };
  namespace literals {
    operator "X"(char const*);
  }
}
using namespace A::literals;

// File b.h:
namespace B {
  class ExtNum { … };
  namespace literals {
    operator "X"(char const*);
```

```
  }
}
using namespace B::literals;

// File main.c:
#include "a.h"
#include "b.h"  // No conflict.
X xx = 123X;  // Error: Ambiguous.
```

4 Use cases

These examples will illustrate the use of user-defined literal with the proposed mechanism. We will usually start with a declaration, and the actual initialization. We have also tried to preserve integration with other proposals, specifically Lawrence Crowl's separator proposal [n2281].

The **b**, **s**, and **df** suffixes are obvious candidates for the standard library.

4.1 Binary literals

This is a frequently requested feature:

```
unsigned long long operator"b" (const char*);
int b32 = 11101111101101110001111111011b;
```

People who propose binary literals usually combine their proposal with a proposal for some kind of separators to heal readability. Combining this proposal with [n2281], we would get:

```
int b32 = 1110_1111_1011_0111_0001_1111_1101_1011b;
```

Note that use of _ as a separator does not clash with our use of _ as a suffix character as long as the chosen suffix characters are not in the set of "digits" being separated.

4.2 Hexadecimal literals

We have hexadecimal literals using the 0x prefix. Where we use hexadecimal literals, we must avoid choosing a first suffix character from the hexadecimal "digit" set [abcdefABCDEF]. Otherwise Maximal Munch will swallow the suffix. For example:

```
int operator "cm" (unsigned long long);     // centimeters
int x = 12345cm;        // fine
int y = 0xascdefcm;     // oops: unknown suffix 'm'
```

4.3 String literals

Literals of type **`std::string`** (as opposed to **`const char*`**) are frequently requested:

```
string operator"s" (const char* p, size_t len)
{
        return string = string(p, len);
```

```
}

string mystring = "Hello World"s;

string cat = "hello"s+ ' ' + "world";
```

4.4 Decimal floating point literals

This is a key motivating case. C has **df** suffixed literals for decimal floating point. C++ does not:

```
Decimal32 operator"df" (const char*);
```

Now we can read C literals directly.

```
Decimal32 d32 = 1.2df;
```

4.5 Raw string literals

We would also like to see raw string literals of type **std::string**:

```
string operator"s" (const char* p, size_t len)
{
        return string = string(p, len);
}

R"[abc\abc]"s      // operator"s"("abc\\abc",7)

"abc\abc"s         // operator"s"("abc\abc",6)
```

Note that to preserve the semantics of literals, literal strings are always "cooked" before they are passed to this conversion operator. In particular, the raw string above passes one more character (the backslash) to the conversion operator than the "ordinary string" (in which **\a** becomes a single character). The quotes are not passed to the conversion function (only the quoted characters), so a conversion function cannot know whether the original string was raw or not.

4.6 Internationalization

Consider strings that need to be converted using a translation table. We could have a **_i18n** (for "internationalization") suffix for literals of such a type. For a string, the **_i18n** conversion operator would look up the string in a translation table tied to the current locale. For a numeric literal, it would produce a string using the current locale's numeric/punctuation properties. For example, **1000000_i18n** may construct—at run time—**str::string("1,000,000")**, whereas **"Hello"_i18n** may look up **"Hello"** in a translation table to produce **std::string("Hola")** at run time.

Such an effect can be achieved with declarations as follows:

```
string operator"_i18n"(const char* str, size_t n) {
  // Look up the string and return the
  // translation.
}

string operator"_i18n"(const char*) {
  // Reformat the numeric literal and return
  // it in string form.
}
```

5 Proposed Wording

Modify the leading grammar rules of 2.4 lex.pptoken as indicated:

preprocessing-token:
header-name
identifier
pp-number
character-literal
user-defined-character-literal
string-literal
user-defined-string-literal
preprocessing-op-or-punc
each non-white-space character that cannot be one of the above

Modify 2.4 lex.pptoken paragraph 2 as indicated:

2 [...] The categories of preprocessing token are: *header names*, *identifiers*, *preprocessing numbers*, *character literals* (including user-defined *character literals*), *string literals* (including user-defined *string literals*), *preprocessing-op-or-punc*, and single non-white-space characters that do not lexically match the other preprocessing token categories. [...]

Modify the leading grammar rules of 2.9 lex.ppnumber as indicated:

pp-number:
digit
. *digit*
pp-number digit
pp-number identifier-nondigit
pp-number **e** *sign*
pp-number **E** *sign*
pp-number **.**

Modify 2.13 lex.literal paragraph 1 as indicated:

1 There are several kinds of literals.20)

literal:
integer-literal
character-literal
floating-literal
string-literal
boolean-literal
user-defined-literal

Add a new section 2.13.7 lex.ext (no underlining to indicate insertion):

user-defined-literal:
user-defined-integer-literal
user-defined-floating-literal
user-defined-string-literal
user-defined-character-literal

user-defined-integer-literal:
decimal-literal ud-suffix
octal-literal ud-suffix
hexadecimal-literal ud-suffix

user-defined-floating-literal:
fractional-constant exponent-part$_{opt}$ *ud-suffix*
digit-sequence exponent-part ud-suffix

user-defined-string-literal:
string-literal ud-suffix

user-defined-character-literal:
character-literal ud-suffix

ud-suffix:
identifier

If a token matches both *user-defined-literal* and another literal kind, then it is treated as the latter. [*Example:* `123_km`, `1.2LL`, `"Hello"s` are all *user-defined-literal*s, but `12LL` is an *integer-literal*. *—end example*]

1 A *user-defined-literal* is treated as a call to a literal operator or literal operator template (13.5.8 over.literal). To determine the form of this call for a given *user-defined-literal L* with *ud-suffix X*, the *literal-operator-id* whose *identifier* is *X* is looked up in the context of *L* using the rules for unqualified name lookup (3.4.1

basic.lookup.unqual). Let *S* be the set of declarations found by this lookup. *S* shall not be empty.

2 If *L* is a *user-defined-integer-literal*, let *n* be the literal without its *ud-suffix*. If *S* contains a literal operator with parameter type `unsigned long long`, the literal *L* is treated as a call of the form

`operator "`*X*`"(`*n*`ULL)`

Otherwise, *S* shall contain a raw literal operator or a literal operator template (13.5.8 over.literal), but not both. If *S* contains a raw literal operator the *literal L* is treated as a call of the form

`operator "`*X*`"("`*n*`")`

Otherwise (*S* contains a literal operator template), *L* is treated as a call of the form

`operator "`*X*`"<'`c_1`', '`c_2`', … , '`c_k`'>()`

where *n* is the source character sequence $c_1c_2...c_k$. [Note: The sequence $c_1c_2...c_k$ can only contain characters from the *basic* source character set. —end note]

3 If *L* is a *user-defined-floating-literal*, let *f* be the literal without its *ud-suffix*. If *S* contains a literal operator with parameter type `long double`, the literal *L* is treated as a call of the form

`operator "`*X*`"(`*f*`L)`

Otherwise, *S* shall contain a raw literal operator or a literal operator template, but not both. If *S* contains a raw literal operator the literal *L* is treated as a call of the form

`operator "`*X*`"("`*n*`")`

Otherwise (*S* contains a literal operator template), *L* is treated as a call of the form

`operator "`*X*`"<'`c_1`', '`c_2`', … , '`c_k`'>()`

where *n* is the source character sequence $c_1c_2...c_k$. [Note: The sequence $c_1c_2...c_k$ can only contain characters from the *basic* source character set. —end note]

4 If *L* is a *user-defined-string-literal*, let *str* be the literal without its *ud-suffix* and let *len* be the number of characters (or code points) in *str* (i.e., its length excluding the terminating null character). The literal *L* is treated as a call of the form

`operator "`*X*`"(`*str, len*`)`

5 If *L* is a *user-defined-character-literal*, let *ch* be the literal without its *ud-suffix*. The literal *L* is treated as a call of the form

`operator "`*X*`"(`*ch*`)`

6 [*Example:*

```
long double operator"x"(long double);

std::string operator"x"(char16_t const*, size_t);

unsigned operator"x"(char const*);

int main() {
  1.2x;     // calls operator"x"(1.2L)
  u"one"x;  // calls operator"x"(u"one", 3)
  12x;      // calls operator"x"("12")
  "two"x;   // Error: No applicable literal operator
}
```

—*end example*]

7 In translation phase 6 (2.1 lex.phases), adjacent string literals are concatenated and *user-defined-string-literal*s are considered string literals for that purpose. During concatenation, *ud-suffix*es are removed and ignored and the concatenation process occurs as described in 2.13.4 (lex.string). At the end of phase 6, if a string literal is the result of a concatenation involving at least one *user-defined-string-literal*, all the participating *user-defined-string-literal*s shall have the same *ud-suffix* and that suffix is applied to the result of the concatenation.

8 [*Example:*

```
int main() {
  L"A" "B" "C"x; // Okay, same as L"ABC"x
  "P"x "Q" "R"y; // Error: two different ud-suffixes
}
```

—*end example*]

Modify 3 basic paragraph 1 as indicated:

7 Two names are the same if
— they are identifiers composed of the same character sequence; or
— they are the names of overloaded operator functions formed with the same operator; or
— they are the names of user-defined conversion functions formed with the same type; or
— they are the names of literal operators (13.5.6 over.literal) formed with the same quoted *identifier*.

Modify 5.1 expr.prim paragraph 1 as indicated:

1 *unqualified-id:*
identifier
operator-function-id
conversion-function-id
literal-operator-id
~ class-name
template-id

Modify 5.1 expr.prim paragraph 7 as indicated:

7 An *identifier* is an *id-expression* provided it has been suitably declared (clause 7). [*Note:* for *operator-function-id*s, see 13.5; for *conversion-function-id*s, see 12.3.2; for *literal-operator-id*s, see 13.5.8 over.literal; for *template-id*s, see 14.2. A *class-name* prefixed by ~ denotes a destructor; see 12.4. Within the definition of a non-static member function, an *identifier* that names a non-static member is transformed to a class member access expression (9.3.1). —*end note*] ...

Add a new section 13.5.8 over.literal (no underlining to indicate insertion):

literal-operator-id:
`operator "`*identifier*`"`

1 A declaration whose *declarator-id* is a *literal-operator-id* shall be a declaration of a namespace-scope function or function template (it could be a friend declaration (11.4 class.friend)), an explicit instantiation or specialization of a function template, or a *using-declaration*s (7.3.3 namespace.udecl). A function declared with a *literal-operator-id* is a *literal operator*. A function template declared with a *literal-operator-id* is a *literal operator template*.

2 The declaration of a *literal operator* must have a *parameter-declaration-clause* equivalent to one of the following:

```
char const*
unsigned long long int
long double
char const*, std::size_t
wchar_t const*, std::size_t
char16_t const*, std::size_t
char32_t const*, std::size_t
```

3 A *raw literal operator* is a literal operator with a single parameter whose type is `char const*` (the first case in the list above).

4 The declaration of a literal operator template must have an empty *parameter-declaration-clause*, and its *template-parameter-list* must have a single *template-parameter* that is a non-type template parameter pack with element type `char`.

5 Literal operators and literal operator templates shall not have C language linkage.

6 [*Note:* Literal operators and literal operator templates are usually invoked implicitly through user-defined literals ([lex.ext] 2.13.6). However, except for the constraints described above, they are ordinary namespace-scope functions and function templates. In particular, they are looked up like ordinary functions and function templates, and they follow the same overload resolution rules. Also, they can be declared `inline` or `constexpr`, they may have internal or external linkage, they can be called explicitly, their address can be taken, etc. —*end note*]

6 Notes

6.1 Library impact

Various components of the standard library and of library TRs can benefit very directly from this proposal. This include <complex>, <string>, and the decimal floating point TR.

However, separate proposals must be made to achieve this.

6.2 Prefixes

The urge to support prefixed user-defined literal was strong especially for strings since that is how we define other types of string literals. However, we dropped the idea because of apparently insurmountable parsing problems. Consider supporting arbitrary prefix and suffix like this:

```
X operator "Pre" "Suf"(char const*);
// Called with "xyz" for token  Pre"xyz"Suf or empty string if none
```

The problem with prefixes is that some keywords can immediately precede literals. For example:

```
and"x" == s
throw"oops"
sizeof"string"
```

Daveed points out (during the Oxford 2007 meeting):

> "I think I found a reason that kills prefixes for user-defined string initializers. Consider the phases of translation. The types of strings need to be determined at phase 5, in order to determine the members of the execution character set, and certainly by phase 6, so that adjacent strings can be sensibly catenated (or diagnosed as "not catenatable"). So when the committee adopts RU or U or u8R or whatever, these have to be hard-wired into the string-catenating logic in phase 6. This looks like

a fundamental contrast to the role of user-defined string "decorations" ... so I guess there is no choice but to go with suffixes. As a very minor consequence, we probably have to apply the user-defined suffix to the string that results from phase-6 catenation."

6.3 Alternative Designs

Several modifications of the constructor syntax were considered and ultimately rejected. (Note: In the following, the character sequence in bold is to be added as additional syntax to the constructor. The exact syntax of these character sequences within a constructor declaration will be discussed later.)

1. Specify just the single suffix or prefix string; e.g.:

   ```
   "df"
   "DQ"
   "utf32"
   ```

   Often that would require writing two otherwise identical constructors.

2. Specify a list of synonymous strings; e.g.: **`"df", "DF"`**. That allows one constructor to handle multiple suffixes or prefixes, but complicates the syntax. Neither of these lets the compiler do any syntax checking for the constructor.

3. Specify a basic typename and the suffix or prefix string(s); e.g.:

   a. **`double "dd", "DD"`**

   b. **`int "long128"`**

4. For user-defined numeric literals, this allows the compiler to check that the syntax matches the specified type except for the suffix, number of digits and exponent range.

   For numeric literals the typename describes the syntax not the size. Typenames **`int`** and **`double`** accept any literal in integer or floating-point syntax with only the specified suffix(es). Typename **`unsigned long`** accepts any integer literal with a **`u`** or **`U`** suffix followed by the specified suffix(es), and **`float`** accepts a floating-point literal with an **`f`** or **`F`** suffix then the specified one(s).

   For user-defined string and character literals it allows the base character type to be specified; e.g.:

   ```
   wchar_t "utf_16"
   ```

5. Instead of a type followed by a quoted string, specify a *literal keyword*. This literal keyword would identify the literals in the C Standard that are known to be missing from the C++ standard. This is somewhat less robust but is easier to describe. For example we can use the keyword **`FLOATING_LITERAL`** to signify the character sequence before the suffix to be a floating literal, and then write the

suffix character sequence after it. (Also, we can omit the quotes in these syntaxes.) For example:

```
FLOATING_LITERAL j
```

6. Specify a *regular expression* describing the type; e.g.:

   ```
   "[0-9]{1-28}long128"
   ```

   That allows much better checking. The extra programming effort is small for the benefit, especially if a sample floating-point regular expression is available.
   This would also allow patterns to match literals like

   ```
   1234d5
   ```

   by accepting a numeric string, **`"d"`**, and another numeric string.

7. Some combination of those. There are good reasons to allow both regular expressions and suffix / prefix strings with optional type names.

8. Using a **`#literal`** syntax that directly gives user string replacement in the lexer which effectively replaces a suffix with the proper constructor call sequence.

6.4 Performance

User-defined literals map onto calls to literal operators. Sometimes such calls will be evaluated at run time. However, the availability of **`constexpr`** functions, and the optional variadic template form of the literal operator allows some important cases to be translated into compile-time constants.

In practice, we expect that the number of user-defined literals requiring execution-time conversion will be relatively small, and that in most cases they will not noticeably affect performance. In any case, existing alternatives also require execution-time conversions.

In the Oxford presentation, there was a great wish to allow user-defined literals to be ROMable to support the embedded system community where there is limited static memory. This design will support that depending on the complexity of the literal construction operator function. However, some user-defined literal construction operator functions cannot be executed at compile time: the literals they construct are not constant expressions and cannot be put into ROM.

The author of a class can and should write appropriate << and >> iostream operators for it, but like any other new class there are restrictions on using **printf ( )** and **scanf ( )**. A type like **Imaginary** can be cast to floating-point and **printf**ed with **"%fj"**, but types like **_Decimal32** have new internal representations so could only be **printf**ed by first converting to a string unless **printf ( )** supported the proposed C **“%HDf” _Decimal32** conversion specifier.

6.5 Related papers

This is compatible with and orthogonal to other proposals including literals for user-defined types [1], generalized initializer lists [2], and braces initialization overloading [4], and benefits from the generalized constant expressions proposal [3].

These other papers primarily propose grouping literals using existing known literals. [1] in particular identifies the possibility of a unique syntax using the **literal** keyword as a constructor, and limits what can be placed inside the constructor so that it can achieve ROMability.

Acknowledgement

We deeply appreciate the email comments from Daveed Vandevoorde and Tom Plum who made key suggestions on the syntax as well as the feedback from David Abrahams, Lawrence Crowl, Francis Glassborow, Michael Spertus, Bill Seymour, and Prem Rao.

References

C++

[1] Bjarne Stroustrup. Literals for user-defined types. N1511

[2] Bjarne Stroustrup and Gabriel Dos Reis. Generalized Initializer Lists. N1509

[3] Gabriel Dos Reis. Generalized Constant Expressions. N1521

[4] Daniel Gutson. Braces Initialization Overloading. N1493

[5] Robert Klarer. Decimal Types for C++. N1839

[6] J. Stephen Adamczyk. Adding the long long type to C++ (Revision 3). N1811

[C++03] ISO/IEC 14882:2003(E), *Programming Language C++*.

[n1892] I. McIntosh, M. Wong, R. Mak. Extensible Literals http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1892.pdf.

[n2281] Lawrence Crowl, Digit Separators http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2281.html.

[n2282] I. McIntosh, M. Wong, R. Mak, R. Klarer. Extensible Literals (revision 2) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2282.pdf.

[n2378] I. McIntosh, M. Wong, R. Mak, R. Klarer. Extensible Literals (revision 3) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2007/n2378.pdf.

C

[7] Extension for the programming language C to support decimal floating-point arithmetic. N1137

[8] The type and representation of unsuffixed floating constant. N1108

[C99] ISO/IEC 9899:1999(E), *Programming Language C*.

Other

[9] Herb Sutter. The New C++: C and C++: Wedding Bells? Oct 2002, C++ User's Journal.

[10] Bjarne Stroustrup. C and C++: Siblings. July 2002, C++ User's journal.

[11] Bjarne Stroustrup. C and C++: A Case for Compatibility. Aug 2002, C++ User's Journal.

[12] Bjarne Stroustrup. The Design and Evolution of C++. Addison-Wesley. 1994.

[13] IEEE 754R - Draft Standard for Floating Point Arithmetic P754/D0.14.2.

[14] IBM C/370 Language Reference Manual.

[15] Cray Architecture Manual.

[16] Bjarne Stroustrup. *Sibling Rivalry: C and C++*. (AT&T Labs — Research Technical Report TD-54MQZY, January 2002), http://www.research.att.com/ ~bs/sibling_rivalry.pdf.

[17] IBM z/Architecture Principles of Operation, http://publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/DZ9ZR003/CCONTENTS?SHELF=DZ9ZBK03&DN=SA22-7832-03&DT=20040504121320.