User-defined Literals
(aka. Extensible Literals (revision 5))

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
klarer@ca.ibm.com
jens.maurer@gmx.net
public@alisdairm.net
bs@cs.tamu.edu
daveed@edg.com

Document number: N2765=08-0275
Date: 2008-09-18
Project: Programming Language C++, Core Working Group
Reply-to: David Vandevoorde (daveed@edg.com)
Revision: 5

Abstract

This paper is Revision 5 of N1892, N2282, N2378, and N2750, 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.

This revision makes a modification to the syntax of literal operators and drops most of the non-wording parts of the earlier papers. It also adds a paragraph in the library wording to reserve suffixes for future use by the standard library.

1 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. [...]

Note to the editor: The Core Working Group suspects that the terms in 2.4 lex.pptoken paragraph 2 should not be italicized.

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 literal suffix 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<'c1', 'c2', … , 'ck'>()`

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<'c1', 'c2', … , 'ck'>()`

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 "" w(long double);

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

unsigned operator "" w(char const*);

int main() {
```

```
  1.2w;      // calls operator "" w(1.2L)
  u"one"w;   // calls operator "" w(u"one", 3)
  12w;       // calls operator "" w("12")
  "two"w;    // 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 (13.5.8 over.literal) formed with the same literal suffix 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 The *identifier* in a *literal-operator-id* is called a *literal suffix identifier*.

2 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*.

3 The declaration of a literal operator shall 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
```

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

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

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

7 [*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*]

8 [*Example:*

```
void operator "" _km(long double);  // Okay
string operator "" _i18n(char const*, size_t);  // Okay
template<char ...>
   int operator "" \u03C0();  // Okay (UCN for lowercase pi)
float operator ""E(char const*);
                // Error: ""E (with no intervening space) is a single token
float operator " " B(char const*);
                // Error: non-adjacent quotes
string operator "" 5X(char const*, size_t);
                // Error: invalid literal suffix identifier
double operator "" _miles(double);
                // Error: invalid parameter-declaration-clause
template<char ...> int operator "" j(char const*);
                // Error: invalid parameter-declaration-clause
```

—*end example*]

Add a new section 17.6.4.3.6 usrlit.suffix as follows.

17.6.4.3.6 User-defined literal suffixes [usrlit.suffix]

1 Literal suffix identifiers that do not start with an underscore are reserved for future standardization.

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.