Extensible Literals (revision 2)

Ian McIntosh, Michael Wong, Raymond Mak, Robert Klarer
ianm@ca.ibm.com
michaelw@ca.ibm.com
rmak@ca.ibm.com
rklarer@ca.ibm.com

Document number: N2282=07-0142
Date: 2007-05-06
Project: Programming Language C++, Evolution Working Group
Reply-to: Michael Wong (michaelw@ca.ibm.com)
Revision: 2

Abstract

This paper is Revision 2 of n1892[n1892] and proposes additional forms of literals using modified syntax and semantics to provide extensible user-defined literals. Extensible literals allow user-defined classes to provide new literal syntaxes and / or data representations, capabilities previously available only for basic types. It increases compatibility with C99 and with future C enhancements as well more flexible C++ literals

An extensible literal translation operator function defines the literal syntax it accepts, and translates that to the data representation. This proposal limits the solution to the suffix form.

The proposal requires a fairly simple change to the core language.

It has been presented since 205 Tremblant meeting and has been greeted with a desire to support this feature, but we have no way of resolving the implementation difficulties. Daveed Vandevoorde offered a solution in Oxford which is essentially presented in this paper.

Revision history:
Revision 2:

- Update on a Syntax suggested by Daveed Vandevoode/Dave Abraham from the April, 2007 Oxford meeting

1 The Problem

C++ recognizes literals for its basic data types. For those, the syntax of the literal (e.g., the presence of a decimal point, exponent or alphabetic suffix) identifies the type. The magnitude further specializes the exact type. 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 suffix dd for a decimal literal) .

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. Defining new literal syntax can be a problem, and providing compatibility with C is a problem.

This paper draws on two basic principles of C++ design:

- User-defined types should have all the same support facilities as built-in types, and currently that facility is not extended to user-defined literals [12]
- C compatibility should be maintained as far as possible and currently, we cannot accept certain C literals [9,10,11,16]

The existing mechanisms work well when existing literals (integers, floating-point and string literals) are suitable. Other proposals [1,2] would 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 modified operator syntax and semantics to provide extensible user-defined literals.

C++ already has extensible data types using classes and templates and overloaded operations on them. What it also needs is extensible literals to match the robustness that extensible types deliver.

2 The Solution: The Basic Idea

The basic idea is that when the compiler lexes tokens, it includes in its search after the literal cases, a extensible-literal token.

```
literal:
            [... existing cases]
            extensible-literal
```

This user-literal token would consists of the following:

```
extensible-literal:
            integer-literal identifier-nondigit
            character-literal identifier-nondigit
            floating-literal nondigit-nondigit
            string-literal identifier-nondigit
            extensible-literal identifier-nondigit
            extensible-literal digit
```

These extensible-literals exclude the normal literal cases such as 1U which remains an integer-literal despite it qualifying as an extensible-literal.

The conversion occurs with a new kind of free function that is an operator based on the suffix of the user-defined literal. We choose a free function to allow extensible literal to be written as needed after the fact and to participate in producing results of built-in types.

3 Goals

The main goal for literal suffixes is to handle every suffix currently in or proposed for C. A second goal is to handle every suffix in common extensions to C. Currently, we are not able to handle string and character literal prefix.

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.

4 Extensible Literal Syntax

An extensible literal is either an *extensible numeric literal* or an *extensible string literal*.

An extensible numeric literal must start with a digit and may contain any characters that would be allowed in an integer or floating-point literal, followed by an alphanumeric *extensible literal suffix* (**"d"** in the example above) accepted by some extensible literal constructor; for example:

1234d
12.34df
12.34e5dd
12.34e-10dq
1_I
12345678901234567890123456789012345678901234567890verylong
12.34.56p

The length may exceed the maximum for basic types, and the character sequence need not match their syntax.

5 Extensible Literal Pattern Syntax

An extensible literal operator function needs to specify the *extensible literal pattern* for the extensible literals it accepts so that matching of the literal can occur. This will occur between the operator-name and the open bracket of the parameter list, in double quotes. The double quotes will identify the suffix pattern, and it will lead to accepting mix cases. For example, a decimal floating point literal can be:

```
Decimal32 operator”df” (char const *);
Imaginary operator”i” (char const *);
```

The appropriate operator is called matching the right suffix and it is passed a character array with the literal in quotation marks minus the suffix. For example, the above operator functions would call operator"i" with

```
{ '1', '.', '2', '\0' }
```

while operator"df" would call it with:

```
{ '1', '2'  '.', '3', '4', '\0' }
```

The syntax can also accept a basic type such as:

```
Length operator"_miles"(float);
```

This would be called when encountering a user-literal whose production started with a floating-point literal (or an integer-literal or character-literal as the case may be). If more then one matching literal conversion operator is found, it is an error.

To facilitate support for compile-time and ROMability, a literal conversion operator can be a constexpr:

```
constexpr Imaginary operator"i"(float x) {
          return Imaginary(x);
}
Imaginary z = 1+2i;
```

Or it could be a variadic template:

```
template<char ... Cs>  // Must be "char ..."
const_expr Imaginary operator"i"();  // Must have no
function-call parameter.

Imaginary z = 1.2i;  // Calls: operator"i"<'1', '.',
'2'>()
```

The choice of a free function allows extensible literal to be written as needed after the fact and to participate in producing results of built-in types. For example:

```
const-expr double operator"pi"(double e) {
     return x*3.14;
}
double circum(double r) {
     return 2pi*r;  // Okay.
}
```

6 Remaining Controversial items

The urge to support prefixed extensible literal was strong especially for strings since that is how we define other types of string literals. This was ultimately dropped due to the complexity involved. It may return if we can see a solution. If a solution could be found, we could support arbitrary prefix and suffix as follows:

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

The problem is that some keywords can immediately precede literals; e.g.:

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

In Daveed opinion in a group email, he pointed out:
“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.”

7 Other options

Several options in 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.:
   double "dd", "DD"
   int "long128"
   For extensible 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 extensible string and character literals it allows the base character type to be specified; e.g.:
   w_char "utf_16"
4. 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.) e.g.:
   FLOATING_LITERAL j
5. 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.
6. Some combination of those. There are good reasons to allow both regular expressions and suffix / prefix strings with optional type names.
7. using a #literal syntax that directly gives user string replacement in the lexer which effectively replaces a suffix with the proper constructor call sequence.

The exact syntax chosen is open for suggestion and is dependent on how much compile-time error checking is desired.

8 Performance

Like static initialization, some extensible literals can be handled at compile time but some would be processed just before **main ( )**. There is no need to require these extensible literals to be processed at compile time, just permit it as an quality of implementation.

The number of extensible literals will be small, the performance difference should not be significant, and existing alternatives would also require execution time conversions, but it would be preferable to completely handle all literals at compile time instead of executing extensible literal constructors at run time.

In fact some extensible literal constructors can be handled at compile time, using the proposed Generalized Constant Expressions. The _Imaginary example above requires only that the compiler float as a constant parameter as a constant expression and that it interprets it to do a conversion it already knows how to do.

9 Restrictions

In the Oxford presentation, there was a great wish to allow extensible literals to be ROMable to support the embedded system community where there are limited static memory. This idea will support that depending on the complexity of the literal construction operator function

But extensible literal construction operator function will not always be executed at compile time, the literals they construct are not constant expressions and cannot be put into ROMs to protect them from modification or for use in embedded systems.

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.

10 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 would be improved by 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 feed back from David Abraham, Bjarne Stroustrup 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] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2005/n1892.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), <www.research.att.com/~bs/sibling_rivalry.pdf>.
[17] IBM z/Architecture Principles of Operation, <publibz.boulder.ibm.com/cgi-bin/bookmgr_OS390/BOOKS/DZ9ZR003/CCONTENTS?SHELF=DZ9ZBK03&DN=SA22-7832-03&DT=20040504121320 >