Reconsidering literal operator templates for strings

Document #:	D0424R0
Date:	2016-08-15
Project:	Programming Language C++
Audience:	Evolution Working Group

Reply-to: Louis Dionne <ldionne.2@gmail.com>

1 Introduction

C++11 added the ability for users to define their own literals suffixes. Several forms of literal operators are available, but one notable omission is the literal operator template for string literals:

```
template <typename CharT, CharT ...chars>
auto operator"" _udl(); // invalid right now

auto x = "abcd"_udl; // would call operator""_udl<char, 'a', 'b', 'c', 'd'>
```

Furthermore, [N3599] tried to make this right in 2013 by proposing to add the missing literal operator, but the paper was rejected at that time with the following conclusion ([CWG66]):

> Revise with additional machinery for compile time string processing

This short paper argues that the literal operator template for string literals should be added as-is to C++, and that no additional machinery for compile-time string processing should be required for its acceptance.

2 Motivation

There are three main reasons why the author thinks [N3599] should be accepted without requiring additional compile-time string processing machinery.

First, several use cases for the literal operator template do not require any sort of compile-time string processing, so the feature would be useful on its own. Indeed, after writing the [Boost.Hana] metaprogramming library, several metaprogramming-inclined individuals have contacted the author of this paper with problems that were solvable with the literal operator template; none of them required additional *string processing* machinery that would be suitable for inclusion in the standard. These problems would benefit from having better support for metaprogramming in the standard, but that is a distinct topic. A simple example of this is implementing named parameters:

```
template <char ...s>
struct argument {
  template <typename T>
```

```
  std::pair<argument<s...>, T> operator=(T value) {
    return {{}, value};
  }

  template <typename T, typename ...Args>
  auto operator()(std::pair<argument<s...>, T> a, Args ...) const {
    return a.second;
  }

  template <typename Arg, typename ...Args>
  auto operator()(Arg, Args ...args) const {
    static_assert(sizeof...(Args) != 0, "missing argument");
    return (*this)(args...);
  }
};

template <typename CharT, CharT ...s>
argument<s...> operator"" _arg() { return {}; }

template <typename ...Args>
void create_window(Args ...args) {
  int x = "x"_arg(args...);
  int y = "y"_arg(args...);
  int width = "width"_arg(args...);
  int height = "height"_arg(args...);
  // etc...
}

int main() {
  create_window("x"_arg = 20, "y"_arg = 50, "width"_arg = 100, "height"_arg = 5);
}
```

Note that this is for illustration only. In a real implementation, one would be wary of not copying the arguments unless necessary, marking functions `constexpr` and `noexcept` whenever possible, etc...

Here, no string processing whatsoever is required. Also, while finding the first `pair` of the parameter pack whose first element represents the right argument would benefit from additional machinery, the author believes that to be completely orthogonal to the current proposal. Also of interest is to note that the example above uses compile-time strings solely to associate a unique tag to a value, a pattern which arises very commonly in C++ metaprogramming. Thus, use cases abound and the proposed feature could for instance be used to eliminate the explicit declaration of tags in the following code:

```
namespace tags { struct name; struct age; } // need to declare tags beforehand

BOOST_FUSION_DEFINE_ASSOC_STRUCT(
```

```
  /* global scope */, Person,
  (std::string, name, tags::name)
  (int, age, tags::age)
)

int main() {
    Person john{"John", 30};
    std::string name = at_key<tags::name>(john);
    int age = at_key<tags::age>(john);
}
```

Indeed, with the proposed feature, the above code can become

```
struct Person {
  BOOST_HANA_DEFINE_STRUCT(Person,
    (std::string, name),
    (int, age)
  );
};

int main() {
  Person john{"John", 30};
  std::string name = at_key(john, "name"_s);
  int age = at_key(john, "age"_s);
}
```

In situations where the number of tags is large, separate tag declarations can be very tedious to keep in sync with the original structure. Instead, it is much better if the library is able to relate the `name` in the structure definition to the `name` used in the `main` function without an additional tag declaration. Even better would be proper support for static reflection, but this is beyond the scope of this paper. Also, not all problems where the proposed feature would be useful can be solved with static reflection, so one is not a substitute for the other.

Secondly, the proposed feature is already implemented in Clang and GCC, and it is getting wider and wider usage. The author assumes the feature to be easy to support for implementations, and thinks that it should be standardized instead of letting its availability vary across compilers. This would make the life of users easier instead of forcing them to use ugly workarounds such as the following to comply with the standard:

```
template <char ...s>
struct argument {
  // ... as before ...
};

template <std::size_t ...N, typename Literal>
argument<Literal::get(N)...> make_argument(std::index_sequence<N...>, Literal) {
  return {};
}
```

```
#define MAKE_ARGUMENT(LITERAL)                                          \
  make_argument(std::make_index_sequence<sizeof(LITERAL)-1>{}, []{      \
    struct Literal {                                                    \
      static constexpr char get(int i) { return LITERAL[i]; }           \
    };                                                                  \
    return Literal{};                                                   \
  }())
/**/

// ... as before ...

int main() {
  create_window(MAKE_ARGUMENT("x") = 20, MAKE_ARGUMENT("y") = 50,
                MAKE_ARGUMENT("width") = 100, MAKE_ARGUMENT("height") = 5);
}
```

Finally, defining string processing machinery is so non-trivial that even runtime C++ lacks any useful form of it. The author thinks that dismissing the proposed feature for lack of such machinery is unwise, since it blocks a lot of use cases that are very basic in their string processing requirements for the benefit of a few more advanced use cases. Furthermore, going forward with this feature does not block us in any way from adding more advanced machinery to the language later on. In fact, it may very well allow us to gain more experience with what kind of machinery would be useful.

3 Proposed Wording

The wording initially proposed in [N3599] still applies to the current draft ([N4606]), and so it should be used. For convenience, it is copied here from the original paper:

The term of art *literal operator template* is split into two terms, *numeric literal operator template* and *string literal operator template*. The term *literal operator template* is retained and refers to either form.

Replace "literal operator template" with "numeric literal operator template" in **[lex.ext] (2.14.8)/3** and **[lex.ext] (2.14.8)/4**:

> [...] Otherwise, `S` shall contain a raw literal operator or a numeric literal operator template (13.5.8), but not both. [...] Otherwise (`S` contains a numeric literal operator template), `L` is treated as a call of the form [...]

Change in **[lex.ext] (2.14.8)/5**:

> If L is a *user-defined-string-literal*, let C be the element type of the string literal as determined by its *encoding-prefix*, let *str* be the literal without its *ud-suffix*, and let *len* be the number of code units in *str* (i.e., its length excluding the terminating null character). If S contains a literal operator with parameter types `const C*` and `std::size_t`, the ~~The~~ literal L is treated as a call of the form `operator"" X(str, len)`. Otherwise, S

shall contain a string literal operator template (13.5.8), and L is treated as a call of the form `operator""X<C,e's'1,e's'2,...,e's'k>()` where *e* is empty when the *encoding-prefix* is `u8` and is otherwise the *encoding-prefix* of the string literal, and *str* contains the sequence of code units $s_1 s_2 ... s_k$ (excluding the terminating null character).

Change in **[over.literal] (13.5.8)/5**:

~~The declaration of a literal operator template shall have an empty *parameter-declaration-clause* and its *template-parameter-list* shall have~~ A *numeric literal operator template* is a literal operator template whose *template-parameter-list* has a single *template-parameter* that is a non-type template parameter pack (14.5.3) with element type `char`. A *string literal operator template* is a literal operator template whose *template-parameter-list* comprises a type *template-parameter* C followed by a non-type template parameter pack with element type C. The declaration of a literal operator template shall have an empty *parameter-declaration-clause* and shall declare either a numeric literal operator template or a string literal operator template.

4 Acknowledgements

Thanks to Richard Smith for doing all the hard work in [N3599].

5 References

[N3599] Richard Smith, Literal operator templates for strings
http://open-std.org/JTC1/SC22/WG21/docs/papers/2013/n3599.html

[CWG66] Richard Smith, EWG Issue #66
http://cplusplus.github.io/EWG/ewg-active.html#66

[Boost.Hana] Louis Dionne, Boost.Hana, A modern metaprogramming library
https://github.com/boostorg/hana

[N4606] Richard Smith, Working Draft, Standard for Programming Language C++
https://github.com/cplusplus/draft/blob/master/papers/n4606.pdf