Document Number: P0330R1
Date: 2017-10-12
Revises: P0330R0
Reply-to: Rein Halbersma <rhalbersma at gmail dot com>
Audience: LWG

User-Defined Literals for `size_t`

1 Introduction

We propose the user-defined suffix `zu` for `size_t` literals. This allows the succinct and convenient left-to-right `auto` variable initialization:

```
auto s = 0zu; // local variable s has value 0 and type size_t
```

2 Motivation and Scope

2.1 The main motivations for this proposal are:

- `int` is the default type deduced from integer literals without suffix;
- `size_t` is almost unavoidable when using the standard containers' element access or `size()` member functions;
- comparisons and arithmetic with integer types of mixed signs or different conversion ranks can lead to surprises;
- surprises range from (pedantic) compiler warnings to undefined behavior;
- using existing unsigned integer literals (such as `ul`) is not a general solution;
- explicit typing or `static_cast` are rather verbose;
- a user-defined suffix for `size_t` literals is a succinct and convenient way to express coding intent.

2.2 The proposed naming of the literal suffix `zu` was motivated by the `%zu` length modifier for `size_t` formatted I/O in the C standard library header `<stdio.h>`. See 7.21.6.1/7 for `fprintf` and 7.21.6.2/11 `fscanf`, numbered relative to WG14/N1539 (see sections 4.1 and 4.2 for a discussion of this design decision and possible alternative namings):

```
printf("%zu", 0zu); // prints 0
```

2.3 The scope of this proposal is limited to adding a literal suffix `zu` for the support type `size_t` defined in the Standard Library header `<cstddef>`, and also making this suffix available through the headers `<cstdio>`, `<cstdlib>`, `<cstring>`, `<ctime>`, and `<cwchar>`. See section 4.3 for a discussion of this design decision and section 7.2 for the proposed wording.

2.4 A previous version of this proposal (WG21/N4254) also proposed adding a user-defined suffix for literals of the type `ptrdiff_t` defined in `<cstddef>`. This part of the proposal has been dropped based on feedback from LEWG.

2.5 Note that a technically similar proposal could be made for the fixed-width integer types in the Standard Library header `<cstdint>`, such as user-defined suffixes `uX` for literals of type `uintX_t`, with `X` running over `{ 8, 16, 32, 64 }`. However, these types do not arise naturally when using the standard containers or algorithms. Furthermore, this would require a more thorough analysis of a good naming scheme that is both brief, intuitive, and without name clashes with other user-defined literals in the Standard Library. We therefore do not propose to add user-defined suffixes for these types.

2.6 For historical reference, see the earlier discussion on std-proposals.

3 Extended Example

3.1 As an illustrative example enabled by this proposal, consider looping over a `vector` and accessing both the loop index `i` as well as the vector elements `v[i]`

```
#include <cstddef> // or <cstdio>, <cstdlib>, <cstring>, <ctime>, <cwchar>
#include <vector>
using namespace std::support_literals;

int main() {
  auto v = std::vector<int> { 98, 03, 11, 14, 17 };

  // loop counter of type size_t, initialized to 0
  for (auto i = 0zu, s = v.size(); i < s; ++i) {
    /* use both i and v[i] */
  }
}
```

This coding style caches the vector's size, similar to the `end()` iterator's caching in a range-based `for` statement. This also fits nicely with the left-to-right `auto` variable initialization, as recommended in Effective Modern C++, Item 5 and GotW #94.

3.2 In the (rare) event that the container's `size_type` is not equal to `size_t` (e.g. because of an exotic user-defined allocator), compilation will simply fail, so that no code will break *silently*. Under these circumstances (as well as in fully generic code), one has to fall back to the more verbose explicit typing

```
// fall back to explicit typing when container::size_type != size_t
for (auto i = decltype(v.size()){0}, s = v.size(); i < s; ++i) { /* ... */ }
```

3.3 A loop counter of type `int` gives the most succinct code, but is likely to lead to sign-related compiler warnings (except for non-standard containers such

as QVector for which the `size()` member function returns `int`), or even to undefined behavior from signed integer overflow

```
// might lead to compiler warnings and signed integer overflow
for (auto i = 0; i < v.size(); ++i) {     // -Wsign-compare
  std::cout << i << ": " << v[i] << '\n'; // -Wsign-conversion
}
```

The above code triggers compiler warnings (shown for Clang and g++). Admittedly, those warnings are rather stringent. But they are not, in general, harmless. Furthermore, in many companies, developers are not free to adjust project-wide mandatory warning levels. But more importantly, even when all compilers warnings have been suppressed, the above loop might (for very large containers) lead to signed integer overflow (which is undefined behavior).

3.4 The example in section 3.3 makes it clear that counters in loops over standard containers should be of unsigned integral type. Note that `[support.types]/7` recommends that implementations choose types for `size_t` whose integer conversion ranks are no greater than that of `signed long int` unless a larger size is necessary to contain all the possible values. This makes `unsigned long` loop counters (which have the recommended maximum conversion rank and which can use the suffix `ul`) a seemingly viable alternative to section 3.1

```
// not guaranteed to be equivalent to section 3.1
for (auto i = 0ul, s = v.size(); i < s; ++i) { /* ... */ }
```

Note, however, because `[support.types]/6` leaves `size_t` an *implementation-defined* unsigned integer type, it is not guaranteed that `unsigned long` (or `unsigned long long` for that matter) is of the same type as `size_t`. Moreover, a user-defined suffix for `size_t` literals also expresses coding intent, and therefore increases code readability and maintainability.

3.5 A fully equivalent alternative to section 3.1 is to name the type of the loop index

```
// equivalent to section 3.1, but more verbose
for (auto i = std::size_t{0}, s = v.size(); i < s; ++i) { /* ... */ }
```

This works under the same circumstances as this proposal (with a fallback to `decltype(v.size())` for exotic containers or fully generic code). Its main drawback is that it is more verbose, especially if the equivalent `static_cast<std::size_t>(0)` were to be employed.

3.6 As an aside, note that the above extended example is not meant to imply a definitive coding style for all index-based `for` loops. E.g., this particular example might be improved by a range-based `for` statement that emits a `size_t` index deduced from a hypothetical zero-based `integral_range` object initialized to `v.size()`

```
// integral_range not actually proposed here, loop over [ 0, v.size() )
```

```
for (auto i : integral_range(v.size()) { /* ... */ }
```

However, for non-zero-based integer ranges (e.g. when skipping the first few elements), the same type deduction issues would reappear, and it would become convenient to write

```
// integral_range not actually proposed here, loop over [ 1, v.size() )
for (auto i : integral_range(1zu, v.size()) { /* ... */ }
```

Regardless of the benefits of such a hypothetical range-based approach for indexed `for` loops, we therefore argue that a user-defined suffix for `size_t` literals has its own merits. Note this proposal does not *enforce* the use of `size_t` literals, it merely *enables* (and perhaps *encourages*) them.

4 Design Decisions

4.1 A previous version of this paper (WG21/N4254) proposed the shorter suffix `z`. Based on feedback from LEWG, this has been changed to `zu`. The main rationale for this change is that `z` alone is not the entirety of what is needed in C I/O formatting. The rather strong consensus was not to use only the modifier `z`, but to use the complete form `zu`.

4.2 For purposes of bikeshedding, we note that other suffixes than the proposed `zu` that contain the letter `z` would also not conflict with existing literals (see section 5 for a full survey). A viable alternative might be to use e.g. the suffix `sz` for `size_t` literals. This loses the congruence with the C I/O length modifier `%zu`, but `sz` is perhaps easier to remember as a mnemonic for `size_t`.

4.3 Note that because other standard headers (`<cstdio>`, `<cstdlib>`, `<cstring>`, `<ctime>` and `<cwchar>`) also define `size_t`, we propose that these headers also make the user-defined suffix `zu` available. In section 7.2, we use wording similar to that of `[iterator.container]/1` that makes the `<iterator>` header available through inclusion of any of the containers, strings or regular expressions headers.

4.4 This proposal follows the existing practice established in WG21/N3642 with respect to the `constexpr` (present) and `noexcept` (absent) specifiers, as well as the use of an appropriately named `inline namespace std::literals::support_literals`.

4.5 There are no decisions left up to implementers, because the proposed wording (see section 7) forms a full specification.

5 Survey of Existing Literal Suffixes

5.1 The literal suffixes for builtin integer types are described in Table 5 of `[lex.icon]/2`. These suffixes (`u` or `U` optionally followed by either `l` or `L` or

by either `ll` or `LL`) do not contain the letter `z` and do not conflict with our proposal.

5.2 The literal suffixes for builtin floating types are described in `[lex.fcon]/1`. These suffixes (one of `f`, `l`, `F`, `L`) do not contain the letter `z` and do not conflict with our proposal.

5.3 The Standard Library header `<chrono>` contains user-defined suffixes for time duration literals, specified in `[time.duration.literals]`. The suffixes currently in use (`h`, `min`, `s`, `ms`, `us`, `ns`) do not contain the letter `z` and do not conflict with our proposal.

5.4 The Standard Library header `<complex>` contains user-defined suffixes for complex number literals, specified in `[complex.literals]`. The suffixes currently in use (`il`, `i`, `if`) do not contain the letter `z` and do not conflict with our proposal.

5.5 The Standard Library header `<string>` contains user-defined suffixes for string literals, specified in `[basic.string.literals]`. The suffix currently in use (`s`) does not contain the letter `z` and does not conflict with our proposal.

5.6 The Standard Library header `<string_view>` contains user-defined suffixes for string view literals, specified in `[string.view.literals]`. The suffix currently in use (`sv`) does not contain the letter `z` and does not conflict with our proposal.

5.7 The Technical Report WG21/N3871 proposes user-defined literals for decimal floating-point literals. The proposed suffixes (`DF`, `DD`, `DL`, `df`, `dd`, `dl`) do not contain the letter `z` and do not conflict with our proposal.

5.8 The Graphics Technical Specification WG21/P0267R0 proposes user-defined suffixes for `double` literals. The proposed suffixes (`ubyte`, `unorm`) do not contain the letter `z` and do not conflict with our proposal.

5.9 The proposal WG21/P0373R0 proposes user-defined file literals prefixing a file's source path. The proposed prefix (`EF`, with `E` a file-encoding prefix such as `b` or `t` for binary or text mode) does not contain the letter `z` and does not conflict with our proposal.

5.10 The Meta library defines a `_z` suffix for `std::integral_constant<size_t, N>` literals using the `template <char...> operator ""` overload (with `N` computed at compile-time from the template parameter pack).

Even though Meta is a support library used in the reference implementation of the proposed Ranges Technical Specification WG21/N4569, the Ranges TS does not rely on Meta's user-defined suffix `_z`. In fact, Meta itself does not even use `_z` internally. We therefore do not anticipate a conflict with our proposal.

5.11 Boost.Hana exposes user-defined suffixes (`_c`, `_s`) for integral constant and compiletime string literals, and also internally defines user-defined literals (`_st`,

`_nd`, `_rd`, `_th`) for tuple indexing. The suffixes in use do not conflict with our proposal.

5.12 Boost.Multiprecision exposes user-defined suffixes (`_cppi`, `_cppui`, `_cppiN`, `_cppuiN`, with `N` an integral power of two) for high precision number literals. The suffixes in use do not conflict with our proposal.

5.13 To the best of our knowledge, other than the aforementioned standard library headers, Technical Specifications and popular open source libraries, there are no other popular user-defined literals that would conflict with our proposal.

6 Impact on the Standard

6.1 This proposal does not depend on other library components, and nothing depends on it. It is a pure library extension, but does require additions (though no modifications) to the standard header `<cstddef>`, (see section 7.1), and also exposing those additions through the headers `<cstdio>`, `<cstdlib>`, `<cstring>`, `<ctime>`, and `<cwchar>` (see section 7.2).

6.2 This proposal can be implemented using C++14 compilers and libraries, and it does not require language or library features that are not part of C++14. In fact, this proposal is entirely implementable using only C++11 language features.

6.3 The consequences of adopting the proposed literal suffix `zu` into the Standard are that both novices and occasional programmers, as well as experienced library implementors, can use left-to-right `auto` variable initializations with `size_t` literals, without having to define their own literal suffix with leading underscore `_zu` in order to do so.

Note that other existing or future Standard Library types (e.g. `chrono::duration` or `complex`) are prevented from adopting the same literal suffix, unless they use overloads of the corresponding `operator ""` that take arguments other than `unsigned long long` (because `[lex.ext]/3` gives these overloads lower precedence during overload resolution).

6.4 There are no (anticipated) conflicts with other literal suffixes, either for builtin types, in other (proposed) Standard Library types, the various Technical Specifications, or in popular open source libraries such as Boost (see section 5).

Note that `[usrlit.suffix]/1` states that literal suffix identifiers that do not start with an underscore are reserved for future standardization. This means that even if there were a popular open source library with a user-defined suffix `_zu`, there would only be a possible conflict with our proposed `zu` suffix for `size_t` literals if that suffix from a third-party library would also be accepted for standardization.

6.5 There are, however, three active CWG issues (cwg#1266, cwg#1620 and cwg#1735) that could impact this proposal. All three issues note that in im-

plementations with extended integer types, the decimal-literal in a user-defined integer literal might be too large for an `unsigned long long` to represent. Suggestions (but no formal proposals) were made to either fall back to a raw literal operator or a literal operator template, or to allow a parameter of an extended integer type. The latter suggestion would be easiest to incorporate into this proposal.

6.6 There is a reference implementation and small test suite available on GitHub for inspection. Note that the reference implementation uses `namespace xstd` and underscored suffix `_zu` because of the restriction from `[lex.ext]/10` that a program containing a user-defined suffix without an underscore is ill-formed, no diagnostic required.

6.7 This proposal successfully compiles and runs on g++ >= 4.7.3, clang >= 3.1 and Visual C++ >= 2015 (possibly on earlier versions of Visual C++ if `constexpr` literals are not used).

7 Proposed Wording

All wording is relative to the Working Draft WG21/N4687.

7.1 Insert in subclause `[cstddef.syn]` in the synopsis of header `<cstddef>` at the appropriate place the namespace `std::literals::support_literals`:

```
namespace std {
  inline namespace literals {
    inline namespace support_literals {
      constexpr size_t operator "" zu(unsigned long long);
    }
  }
}
```

7.2 Insert a new subclause `[support.literals]` between `[support.types]` and `[support.limits]` as follows:

> **21.3 Suffixes for support types [support.literals]**
>
> 1 This section describes a literal suffix for constructing `size_t` literals. The suffix `zu` creates values of type `size_t`.
>
> `constexpr size_t operator "" zu(unsigned long long u);`
>
> 2 Returns: `static_cast<size_t>(u)`.
>
> 3 In addition to being available via inclusion of the `<cstddef>` header, this literal operator is available when any of the following headers is included: `<cstdio>`, `<cstdlib>`, `<cstring>`, `<ctime>`, and `<cwchar>`.

8 Acknowledgments

We gratefully acknowledge Walter E. Brown for acting as our *locum* in committee meetings and for his valuable feedback. We also acknowledge feedback on a previous version of this proposal from Jerry Coffin and Andy Prowl on `<Lounge C++>`, guidance from Daniel Krügler, as well as input from various participants on `std-proposals`.

9 References

`[Boost.Hana]` Louis Dionne: *A modern C++ metaprogramming library* http://www.boost.org/doc/libs/1_61_0/libs/hana/doc/html/namespaceboost__1_1hana__1_1literals.html

`[Boost.Multiprecision]` John Maddock and Christopher Kormanyos: *Extended precision arithmetic types for floating point, integer and rational arithmetic* http://www.boost.org/doc/libs/1_61_0/libs/multiprecision/doc/html/boost_multiprecision/tut/lits.html

`[Effective Modern C++]` Scott Meyers: *42 Specific Ways to Improve Your Use of C++11 and C++14* (*Item 5: Prefer auto to explicit type declarations.*) http://shop.oreilly.com/product/0636920033707.do

`[GotW #94]` Herb Sutter: *AAA Style (Almost Always Auto)* http://herbsutter.com/2013/08/12/gotw-94-solution-aaa-style-almost-always-auto/

`[Meta]` Eric Niebler: *A tiny metaprogramming library* https://ericniebler.github.io/meta/group___integral.html#gaddea0d053893b5bec6ba3d75af70624e

`[N3642]` Peter Sommerlad: *User-defined Literals for Standard Library Types (part 1 - version 4)* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3642.pdf

`[N3871]` Dietmar Kühl: *Proposal to Add Decimal Floating Point Support to C++ (revision 2)* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3871.html

`[N4254]` Rein Halbersma: *User-defined Literals for size_t* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n4254.html

`[N4569]` Eric Niebler: *Working Draft, C++ Extensions for Ranges* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/n4569.pdf

`[N4687]` Richard Smith: *Working Draft, Standard for Programming Language C++* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/n4687.pdf

`[P0276R0]` Michael B. McLaughlin, Herb Sutter and Jason Zink: *A Proposal to Add 2D Graphics Rendering and Display to C++* http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0267r0.pdf

[P0276R0] Andrew Tomazos: *Proposal of File Literals* http://open-std.org/JTC1/SC22/WG21/docs/papers/2016/p0373r0.pdf

[QVector] http://doc.qt.io/qt-5/qvector.html#size

[std-proposals] Morwenn Edrahir: *User defined literal for size_t* https://groups.google.com/a/isocpp.org/forum/#!topic/std-proposals/tGoPjUeHlKo

10 Change History

1. 2014-11-21; Published as N4254.
2. 2016-05-15; Published as P0330R0; summarized LEWG's view re N4254; dropped the proposed suffix for `ptrdiff_t`; changed the proposed suffix for `size_t` to `zu`; added survey of existing literal suffixes.
3. 2017-10-12; Published as P0330R1; expanded the survey of existing literals. Synced the proposed wording with the Working Draft WG21/N4687. Moved the reference implementation from BitBucket to GitHub.