User-defined Literals for Standard Library Types (part 1 - version 4)

Peter Sommerlad

2013-04-18

Document Number:	N3642 (part1 of update of N3531/N3468/N3402)
Date:	2013-04-18
Project:	Programming Language C++

1 History

1.1 Changes from N3531

The following changes (or non-changes) result from the discussion in Bristol.

- Split the proposal to enable voting on imaginary literal suffixes for `std::complex` separately. This is part 1 without the suffixes for `std::complex` imaginary constants. Nevertheless, it includes the complete discussion, because that mainly relates to the previous combined paper. It also contains the complete demo implementation, even for std::complex literals.
- put `const` to the (IMHO wrong position) towards the left to be consistent with the standard's use of syntax.
- rename the parameter `min` of `operator"" min()` to `minutes`.
- Discussion about whether the "Imaginary Literal Suffixes for std::complex" should be `noexcept` as well as `constexpr`. `complex<float>` converts from `double`, and hence can have undefined behavior, i.e., a narrow contract, and hence maybe should not be `noexcept`. Alisdairs suggestion is to advance this paper now and open an issue to look at what should be `noexcept`.

  I conclude no change for `noexcept`. IMHO it is a non-issue for `constexpr` UDL operators, since they are applied by the compiler. Exceptions might only be thrown if called at run-time explicitly, i.e., `operator"" i_f(3.14);` never by using the suffix.

- Change "Effects: creates a complex literal as XXX" to "Returns XXX".
- "[Note: The keyword if is not available as a suffix.]" was criticized by STL for being too helpful, and not consistent with the rest of the standard.
- `min` is not a problem even for platform where its a macro. Some discussion of why "min" instead of "m" for minutes, when we have "us" for microseconds. Comment from Daniel: "s" for "seconds" is consistent with SI units. (I consider that only to be a discussion item indicating no change)
- Discussion about μ (micro) symbol and an `operator"" μs()`. Conclusion to don't put it into the standard, `us` is OK it is an allowed replacement for micro in SI units. remove the note that encourages implementations to provide micro symbol, because that would make user code non-portable.
- in 4.6.1p4, please strike the braces around the example code.
- "since duration suffixes always apply to numbers and string literal suffixes always apply to character array literals". remove always

1.2 Changes from N3468

The following changes were made based on input from BSI.

- move implementation code to appendix.
- add discussion on a suffix for `std::string_ref`.
- refer to SI units abbreviations definition.

1.3 Changes from N3402

The following changes result from discussions in Portland.

- drop binary literals and ask core/evolution first if it would be done by core. it should be done in code via a prefix "`0b`".
- drop real-part `std::complex` literals `operator"" r()` and make the imaginary-part operators `i`, `il`, and `i_f`.
- drop mechanics for type deduction of integers from standard. They should be part of type traits anyway and should be also an integral_constant. This paper still provides their updated implementation as an example for potential implementors.
- make the floating point representation type of `chrono::duration` floating point literals unspecified.

2 Introduction

The standard library is lacking pre-defined user-defined literals, even though the standard reserves names not starting with an underscore for it. Even the sequence of papers that introduced UDL to the standard contained useful examples of suffixes for creating values of standard types such as `s` for `std::string`, `h` for `std::chrono::hours` and `i` for imaginary parts of complex numbers.

Discussion on the reflector in May 2012 and in Portland Oct 2012 showed demand for some or even many pre-defined UDL operators in the standard library, however, there was no consensus how far to go and how to resolve conflicts in naming. One can summarize the requirements of the discussion as follows:

- use an inline namespace for a (group of related) UDL operator(s)
- use an inline namespace `std::literals` for all such UDL namespaces
- ISO units would be nice to have, but some might conflict with existing syntax, such as `F`, `l`, `lm`, `lx`, `''`(seconds) or cannot be represented easily in all fonts, such as Ω or °C.[1]
- suffix `s` was proposed for `std::string` but is also ISO standard for seconds and could be convenient for `std::chrono::duration` values. However, because the overloads for such an `operator"" s()` differ for these two usage scenarios, `s` can also be used for `std::string`.
- an UDL for constructing `std::string` literals should not allocate memory, but use a `string_ref` type, once some like that is available in the standard. Such a facility could easily provide a different suffix to make string literals a `string_ref` instead of a `std::basic_string`.
- any proposal that is made for adding user-defined literal functions to the standard library will evoke some discussion.
- Alberto Ganesh Barbati ¡albertobarbati@gmail.com¿ suggested to provide the number parsing facility to be used by UDL template operators should be exported, so that authors of UDL suffixes could reuse it. Discussion in Portland showed it should be left to the implementer and the general usability of such a feature is limited to the (few) experts implementing UDL template operators. Boost might be a place to provide such a reusable expert-only feature. In addition the `select_int_type` should be an `std::integral_constant` and should be part of the type traits, but under a better name.
- BSI suggested to change suffixes for complex literals to the style of `i_f`, however, the current suffixes where heavily discussed in Portland and the result of the library group decision is used for this paper.

[1] see at http://www.ewh.ieee.org/soc/ias/pub-dept/abbreviation.pdf

- BSI suggested to use suffix s for `std::string_ref` once it is accepted to the standard library. However, I believe we want keep `s` for meaning `std::string` and `std::string_ref` should propose a different suffix such as `sr` for denoting short-hand for converting string literals into type `std::string_ref`. I suggest the `string_ref` proposal adds its own suffix mimicking the approach of this proposal.

Based on this discussion this paper proposes to include UDL operators for the following library components.

- `std::basic_string`, suffix `s` in inline namespace `std::literals::string_literals`
- `std::complex`, suffixes `i, il, i_f` in inline namespace `std::literals::complex_literals`
- `std::chrono::duration`, suffixes `h, min, s, ms, us, ns` in inline namespace `std::literals::chrono_literals`

2.1 Rationale

User-defined literal operators (UDL) are a new features of C++11. However, while the feature is there it is not yet used by the standard library of C++11. The papers introducing UDL already named a few examples where source code could benefit from pre-defined UDL operators in the library, such as imaginary number, or std::string literals.

Fortunately the C++11 standard already reserved UDL names not starting with an underscore '_' for future standardization.

Several library classes representing scalar or numeric types can benefit from pre-defined UDL operators that ease their use: `std::complex` and `std::chrono::duration`. Also `std::basic_string<CharT>` instantiations are a viable candidate for a suffix `operator "" s(CharT const*, size_t)`.

2.2 Open Issues Discussed

2.2.1 Suffixes Utilities - exposition only

It has to be decided if the utilities for implementing UDL suffix operators with integers should be standardized. Discussion in Portland consented in abandoning that for this paper. It also gave consensus to put the binary literals in the hand of the compiler with a prefix of `0b`, e.g., `0b1001` instead.

The template `select_int_type` might be a candidate for the clause [meta.type.synop], aka header `<type_traits>` but with another name.

2.2.2 Upper-case versions of suffixes - dropped

The original version said: "To avoid combinatorial explosion there won't be upper case version of the UDL suffixes unless they are mimicking built-in suffixes." Since we moved

binary to the compiler, this no longer applies for more than one suffix. So we only propose lower case suffixes only, even for `std::complex`.

2.2.3 Suffix r for real-part only std::complex numbers - dropped

It needs to be discussed if this set of suffixes (r, lr, fr, R, LR, FR) for complex numbers with a real part only is actually required and useful. If all viable overloaded versions of constexpr operators are available for std::complex they might not be needed.

Discussion in Portland consented to abandon the `r` suffixes, because they are redundant, once some more operators of `std::complex` will be made `constexpr`.

2.3 Acknowledgements

Acknowledgements go to the original authors of the sequence of papers the lead to inclusion of UDL in the standard and to the participants of the discussion on UDL on the reflector. Special thanks to Daniel Krügler for tremendous feedback on all drafts and to Jonathan Wakely for guidelines on GCC command line options. Thanks to Alberto Ganesh Barbati for feedback on duration representation overflow and suggestion for also providing the number parsing as a standardized library component. Thanks to Bjarne Stroustrup for suggesting to add more rationale to the proposal.

Thanks to all participants in the discussions in groups "library" and "evolution" in Portland.

Thanks to the BSI reviewers of N3468 and Roger Orr as their spokesperson.

Thanks to the library group participants in Bristol discussing the paper N3531 and draft versions of this paper for their feedback and making me aware of it in time.

3 Proposed Library Additions

It must be decided in which section to actually put the proposed changes. I suggest we add them to the corresponding library parts, where appropriate.

3.1 namespace literals for collecting standard UDLs

As a common schema this paper proposes to put all suffixes for user defined literals in separate inline namespaces that are below the inline namespace `std::literals`. [*Note:* This allows a user either to do a `using namespace std::literals;` to import all literal operators from the standard available through header file inclusion, or to use `using namespace std::string_literals;` to just obtain the literals operators for a specific type. — *end note*]

3.2 operator"" s() for basic_string

Make the following additions and changes to library clause 21 [strings] to accommodate the user-defined literal suffix s for string literals resulting in a corresponding string object instead of array of characters.

Insert in 21.3 [string.classes] in the synopsis at the appropriate place the inline namespace `std::literals::string_literals`

```
namespace std{
inline namespace literals{
inline namespace string_literals{
basic_string<char> operator "" s(const char *str, size_t len);
basic_string<wchar_t> operator "" s(const wchar_t *str, size_t len);
basic_string<char16_t> operator "" s(const char16_t *str, size_t len);
basic_string<char32_t> operator "" s(const char32_t *str, size_t len);
}}}
```

Before subclause 21.7 [c.strings] add a new subclause [basic.string.literals]

3.3 Suffix for basic_string literals [basic.string.literals]

```
basic_string<char> operator "" s(const char *str, size_t len);
```

1 *Returns:* `basic_string<char>{str,len}`

```
basic_string<wchar_t> operator "" s(const wchar_t *str, size_t len);
```

2 *Returns:* `basic_string<wchar_t>{str,len}`

```
basic_string<char16_t> operator "" s(const char16_t *str, size_t len);
```

3 *Returns:* `basic_string<char16_t>{str,len}`

```
basic_string<char32_t> operator "" s(const char32_t *str, size_t len);
```

4 *Returns:* `basic_string<char32_t>{str,len}`

[*Note:* The same suffix `s` is used for `chrono::duration` literals denoting seconds but there is no conflict, since duration suffixes apply to numbers and string literal suffixes apply to character array literals. — *end note*]

3.4 Suffixes for chrono::duration values

Make the following additions and changes to library subclause 20.11 [time] to accommodate user-defined literal suffixes for chrono::duration literals.

Insert in subclause 20.11.2 [time.syn] in the synopsis at the appropriate place the inline namespace `std::literals::chrono_literals`.

```
namespace std {
inline namespace literals {
inline namespace chrono_literals{
constexpr
chrono::hours operator"" h(unsigned long long);
constexpr
chrono::duration<unspecified, ratio<3600,1>> operator"" h(long double);
constexpr
chrono::minutes operator"" min(unsigned long long);
constexpr
chrono::duration<unspecified, ratio<60,1>> operator"" min(long double);
constexpr
chrono::seconds operator"" s(unsigned long long);
constexpr
chrono::duration<unspecified> operator"" s(long double);
constexpr
chrono::milliseconds operator"" ms(unsigned long long);
constexpr
chrono::duration<unspecified, milli> operator"" ms(long double);
constexpr
chrono::microseconds operator"" us(unsigned long long);
constexpr
chrono::duration<unspecified, micro> operator"" us(long double);
constexpr
chrono::nanoseconds operator"" ns(unsigned long long);
constexpr
chrono::duration<unspecified, nano> operator"" ns(long double);
}}}
```

Insert in subclause 20.11.5 [time.duration] after subclause 20.11.5.7 [time.duration.cast] a new subclause 20.11.5.8 [time.duration.literals] as follows.

3.4.1 Suffix for duration literals [time.duration.literals]

1 This section describes literal suffixes for constructing duration literals. The suffixes `h,min,s,ms,us,ns` denote duration values of the corresponding types `hours`, `minutes`, `seconds`, `miliseconds`, `microseconds`, and `nanoseconds` respectively if they are applied to integral literals.

2 If any of these suffixes are applied to a floating point literal the result is a `chrono::duration` literal with an unspecified floating point representation.

3 If any of these suffixes are applied to an integer literal and the resulting `chrono::duration` value cannot be represented in the result type because of overflow, the program is ill-formed.

4 [*Example:* The following code shows some duration literals.

```
using namespace std::chrono_literals;
auto constexpr aday=24h;
auto constexpr lesson=45min;
auto constexpr halfanhour=0.5h;
```

— *end example*]

5

```
constexpr
chrono::hours operator"" h(unsigned long long hours);
constexpr
chrono::duration<unspecified, ratio<3600,1>> operator"" h(long double hours);
```

6 *Returns:* A `duration` literal representing `hours` hours.

```
constexpr
chrono::minutes operator"" min(unsigned long long minutes);
constexpr
chrono::duration<unspecified, ratio<60,1>> operator"" min(long double minutes);
```

7 *Returns:* A `duration` literal representing `minutes` minutes.

```
constexpr
chrono::seconds operator"" s(unsigned long long sec);
constexpr
chrono::duration<unspecified> operator"" s(long double sec);
```

8 *Returns:* A `duration` literal representing `sec` seconds.

[*Note:* The same suffix `s` is used for `basic_string` but there is no conflict, since duration suffixes apply to numbers and string literal suffixes apply to character array literals. — *end note*]

```
constexpr
chrono::milliseconds operator"" ms(unsigned long long msec);
constexpr
chrono::duration<unspecified, milli> operator"" ms(long double msec);
```

9 *Returns:* A `duration` literal representing `msec` milliseconds.

```
constexpr
chrono::microseconds operator"" us(unsigned long long usec);
constexpr
chrono::duration<unspecified, micro> operator"" us(long double usec);
```

10 *Returns:* A `duration` literal representing `usec` microseconds.

```
constexpr
chrono::nanoseconds operator"" ns(unsigned long long nsec);
constexpr
chrono::duration<unspecified, nano> operator"" ns(long double nsec);
```

11 *Returns:* A `duration` literal representing `nsec` nanoseconds.

4 Appendix: Possible Implementation

This section shows some possible implementations of the user-defined-literals proposed.

4.1 Compile-time Integer Parsing

For its usage, see the implementation of `std::chrono::duration` literals. This part will not be standardized but it is for exposition of the technique. It is planned to provide this facility through a future Boost library with adapted namespace names.

```
#ifndef SUFFIXESPARSENUMBERS_H_
#define SUFFIXESPARSENUMBERS_H_
#include <cstddef>
namespace std {
namespace parse_int {
template <unsigned base, char... Digits>
struct parse_int{
    static_assert(base<=16u,"only support up to hexadecimal");
    static_assert(! sizeof...(Digits), "invalid integral constant");
    static constexpr unsigned long long value=0;
};

template <char... Digits>
struct base_dispatch;

template <char... Digits>
struct base_dispatch<'0','x',Digits...>{
    static constexpr unsigned long long value=parse_int<16u,Digits...>::value;
};
template <char... Digits>
struct base_dispatch<'0','X',Digits...>{
    static constexpr unsigned long long value=parse_int<16u,Digits...>::value;
};
template <char... Digits>
struct base_dispatch<'0',Digits...>{
    static constexpr unsigned long long value=parse_int<8u,Digits...>::value;
};
template <char... Digits>
struct base_dispatch{
    static constexpr unsigned long long value=parse_int<10u,Digits...>::value;
};

constexpr unsigned long long
pow(unsigned base, size_t to) {
    return to?(to%2?base:1)*pow(base,to/2)*pow(base,to/2):1;
}

template <unsigned base, char... Digits>
```

```
struct parse_int<base,'0',Digits...>{
    static constexpr unsigned long long value{ parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'1',Digits...>{
    static constexpr unsigned long long value{ 1 *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'2',Digits...>{
    static_assert(base>2,"invalid digit");
    static constexpr unsigned long long value{ 2 *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'3',Digits...>{
    static_assert(base>3,"invalid digit");
    static constexpr unsigned long long value{ 3 *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'4',Digits...>{
    static_assert(base>4,"invalid digit");
    static constexpr unsigned long long value{ 4 *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'5',Digits...>{
    static_assert(base>5,"invalid digit");
    static constexpr unsigned long long value{ 5 *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'6',Digits...>{
    static_assert(base>6,"invalid digit");
    static constexpr unsigned long long value{ 6 *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'7',Digits...>{
    static_assert(base>7,"invalid digit");
    static constexpr unsigned long long value{ 7 *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'8',Digits...>{
    static_assert(base>8,"invalid digit");
    static constexpr unsigned long long value{ 8 *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
```

```
};
template <unsigned base, char... Digits>
struct parse_int<base,'9',Digits...>{
    static_assert(base>9,"invalid digit");
    static constexpr unsigned long long value{ 9 *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'a',Digits...>{
    static_assert(base>0xa,"invalid digit");

    static constexpr unsigned long long value{ 0xa *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'b',Digits...>{
    static_assert(base>0xb,"invalid digit");
    static constexpr unsigned long long value{ 0xb *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'c',Digits...>{
    static_assert(base>0xc,"invalid digit");
    static constexpr unsigned long long value{ 0xc *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'d',Digits...>{
    static_assert(base>0xd,"invalid digit");
    static constexpr unsigned long long value{ 0xd *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'e',Digits...>{
    static_assert(base>0xe,"invalid digit");
    static constexpr unsigned long long value{ 0xe *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'f',Digits...>{
    static_assert(base>0xf,"invalid digit");
    static constexpr unsigned long long value{ 0xf *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'A',Digits...>{
    static_assert(base>0xA,"invalid digit");
    static constexpr unsigned long long value{ 0xa *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
```

```
};
template <unsigned base, char... Digits>
struct parse_int<base,'B',Digits...>{
    static_assert(base>0xB,"invalid digit");
    static constexpr unsigned long long value{ 0xb *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'C',Digits...>{
    static_assert(base>0xC,"invalid digit");
    static constexpr unsigned long long value{ 0xc *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'D',Digits...>{
    static_assert(base>0xD,"invalid digit");
    static constexpr unsigned long long value{ 0xd *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'E',Digits...>{
    static_assert(base>0xE,"invalid digit");
    static constexpr unsigned long long value{ 0xe *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
template <unsigned base, char... Digits>
struct parse_int<base,'F',Digits...>{
    static_assert(base>0xF,"invalid digit");
    static constexpr unsigned long long value{ 0xf *pow(base,sizeof...(Digits))
                                               + parse_int<base,Digits...>::value};
};
}}
#endif /* SUFFIXESPARSENUMBERS_H_ */
```

Here comes some example code from my test cases showing the use of that facility:

```
template <char... Digits>
constexpr  unsigned long long
operator"" _ternary(){
    return std::parse_int::parse_int<3,Digits...>::value;
}
constexpr auto five= 012_ternary;
static_assert(five==5, "_ternary should be three-based");
//constexpr auto invalid=3_ternary;

template <char... Digits>
constexpr  unsigned long long
operator"" _testit(){
    return std::suffixes::base_dispatch<Digits...>::value;
}
```

```
constexpr auto a = 123_testit;   // value 123
constexpr auto b = 0123_testit;  // value 0123
constexpr auto c = 0x123_testit; // value 0x123
```

4.2 integral type fitting

During discussion in Portland it was assumed that this facility should be part of the `type_traits` header, because it provides similar facilities. The discussion further provided a simplified implementation of it through inheriting from `std::integral_constant`. For the purpose of this paper it is exposition only for the technique of it.

It is planned to provide it via Boost for others to try and use.

```
#ifndef SELECT_INT_TYPE_H_
#define SELECT_INT_TYPE_H_
#include <type_traits>
#include <limits>

namespace std {
namespace select_int_type {

template <unsigned long long val, typename... INTS>
struct select_int_type;

template <unsigned long long val, typename INTTYPE, typename... INTS>
struct select_int_type<val,INTTYPE,INTS...>
:integral_constant<typename conditional<
(val<=static_cast<unsigned long long>(std::numeric_limits<INTTYPE>::max()))
,INTTYPE
,typename select_int_type<val,INTS...>::value_type >::type , val> {
};

template <unsigned long long val>
struct select_int_type<val>:integral_constant<unsigned long long,val>{
};

}}
#endif /* SELECT_INT_TYPE_H_ */
```

Here are some examples from my test cases to show an example on how to use. For others see the following section on binary literals which we will not standardize as is.

```
using std::select_int_type::select_int_type;
template <unsigned long long val>
constexpr
typename select_int_type<val,
short, int, long long>::value_type
foo() {
    return  select_int_type<val,
```

```
            short, int, long long>::value;
}
static_assert(std::is_same<decltype(foo<100>()),
              short>::value,"foo<100>() is short");
static_assert(std::is_same<decltype(foo<0x10000>()), int>::value,
              "foo<0x10000>() is int");
static_assert(std::is_same<decltype(foo<0x100000000000>()), long long>::value,
              "foo<0x100000000000>() is long long");
```

4.3 binary

This is exposition only and no longer considered to be standardized. A version of it might be provided through Boost as _b and corresponding versions. It is used to demonstrate the possible facilities, but it was considered in Portland that it could easily add to compile times, if binary literals would require to use that facility. Users who can not wait for binary literals to be implemented by their compiler can use it for their needs until then.

```
#ifndef BINARY_H_
#define BINARY_H_
#include <limits>
#include <type_traits>
#include "select_int_type.h"
namespace std{
namespace suffixes{
namespace binary{
namespace __impl{

template <char... Digits>
struct bitsImpl{
        static_assert(! sizeof...(Digits),
                        "binary literal digits must be 0 or 1");
        static constexpr unsigned long long value=0;
};

template <char... Digits>
struct bitsImpl<'0',Digits...>{
        static constexpr unsigned long long value=bitsImpl<Digits...>::value;
};

template <char... Digits>
struct bitsImpl<'1',Digits...>{
        static constexpr unsigned long long value=
                        bitsImpl<Digits...>::value|(1ULL<<sizeof...(Digits));
};
using std::select_int_type::select_int_type;
}
```

```
template <char... Digits>
constexpr typename
__impl::select_int_type<__impl::bitsImpl<Digits...>::value,
      int, unsigned, long, unsigned long, long long>::value_type
operator"" b(){
        return  __impl::select_int_type<__impl::bitsImpl<Digits...>::value,
                        int, unsigned, long, unsigned long, long long>::value;
}
template <char... Digits>
constexpr typename
__impl::select_int_type<__impl::bitsImpl<Digits...>::value,
      long, unsigned long, long long>::value_type
operator"" bl(){
        return  __impl::select_int_type<__impl::bitsImpl<Digits...>::value,
                                long, unsigned long, long long>::value;
}
template <char... Digits>
constexpr auto
operator"" bL() -> decltype(operator "" bl<Digits...>()){
        return  operator "" bl<Digits...>();
}

template <char... Digits>
constexpr typename
__impl::select_int_type<__impl::bitsImpl<Digits...>::value,
       long long>::value_type
operator"" bll(){
        return  __impl::select_int_type<__impl::bitsImpl<Digits...>::value,
                                long long>::value;
}
template <char... Digits>
constexpr auto
operator"" bLL() -> decltype(operator "" bll<Digits...>()){
        return  operator "" bll<Digits...>();
}

template <char... Digits>
constexpr typename
__impl::select_int_type<__impl::bitsImpl<Digits...>::value,
      unsigned, unsigned long>::value_type
operator"" bu(){
        return  __impl::select_int_type<__impl::bitsImpl<Digits...>::value,
                                unsigned, unsigned long>::value;
}

template <char... Digits>
constexpr auto
operator"" bU() -> decltype(operator "" bu<Digits...>()){
        return  operator "" bu<Digits...>();
```

```
}

template <char... Digits>
constexpr typename
__impl::select_int_type<__impl::bitsImpl<Digits...>::value,
       unsigned long>::value_type
operator"" bul(){
        return  __impl::select_int_type<__impl::bitsImpl<Digits...>::value,
                                unsigned long>::value;
}
template <char... Digits>
constexpr auto
operator"" bUL() -> decltype(operator "" bul<Digits...>()){
        return  operator "" bul<Digits...>();
}
template <char... Digits>
constexpr auto
operator"" buL() -> decltype(operator "" bul<Digits...>()){
        return  operator "" bul<Digits...>();
}
template <char... Digits>
constexpr auto
operator"" bUl() -> decltype(operator "" bul<Digits...>()){
        return  operator "" bul<Digits...>();
}
template <char... Digits>
constexpr unsigned long long
operator"" bull(){
        return __impl::bitsImpl<Digits...>::value;
}
template <char... Digits>
constexpr unsigned long long
operator"" bULL(){
        return __impl::bitsImpl<Digits...>::value;
}
template <char... Digits>
constexpr unsigned long long
operator"" buLL(){
        return __impl::bitsImpl<Digits...>::value;
}
template <char... Digits>
constexpr unsigned long long
operator"" bUll(){
        return __impl::bitsImpl<Digits...>::value;
}
} // binary
} //suffixes
} // std
```

```
#endif /* BINARY_H_ */
```

4.4 basic_string

This section is for exposition only to show two possible implementations. The macro version can be briefer, the non-macro version is more elaborate and duplicates code, otherwise they are equivalent.

```
#ifndef STRING_SUFFIX_H_
#define STRING_SUFFIX_H_
#include <string>
namespace std{
inline namespace literals{
inline namespace string_literals{
#if 0
#define __MAKE_SUFFIX_S(CHAR) \
        basic_string<CHAR>\
operator "" s(CHAR const *str, size_t len){\
        return basic_string<CHAR>(str,len);\
}

__MAKE_SUFFIX_S(char)
__MAKE_SUFFIX_S(wchar_t)
__MAKE_SUFFIX_S(char16_t)
__MAKE_SUFFIX_S(char32_t)
#undef __MAKE_SUFFIX
#else // copy-paste version for proposal

basic_string<char>
operator "" s(char const *str, size_t len){
return basic_string<char>(str,len);
}
basic_string<wchar_t>
operator "" s(wchar_t const *str, size_t len){
return basic_string<wchar_t>(str,len);
}
basic_string<char16_t>
operator "" s(char16_t const *str, size_t len){
return basic_string<char16_t>(str,len);
}
basic_string<char32_t>
operator "" s(char32_t const *str, size_t len){
return basic_string<char32_t>(str,len);
}
#endif
}}}
#endif /* STRING_SUFFIX_H_ */
```

Here are some test cases testing compilability of the UDL operators for `std::basic_string`.

```
using namespace std::string_literals;
static_assert(std::is_same<decltype("hallo"s),std::string>{},
              "s means std::string");
static_assert(std::is_same<decltype(u8"hallo"s),std::string>{},
              "u8 s means std::string");
static_assert(std::is_same<decltype(L"hallo"s),std::wstring>{},
              "L s means std::wstring");
static_assert(std::is_same<decltype(u"hallo"s),std::u16string>{},
              "u s means std::u16string");
static_assert(std::is_same<decltype(U"hallo"s),std::u32string>{},
              "U s means std::u32string");

void testStringSuffix(){
        ASSERT_EQUAL(typeid("hi"s).name(),typeid(std::string).name());
        ASSERT_EQUAL(std::string{"hello"},"hello"s);
}
```

4.5 std::complex

This is the example implementation of UDL operators for creating imaginary values of type `std::complex`. Note that this deviates from the original proposal which put the letter indicating the type in front of the i. This caused the problem that you couldn't create a `std::complex<float>` from a hexadecimal integer, as `0x1fi` would have been interpreted as `std::complex<double>0,15` instead of `std::complex<float>0,1`. And because "if" is a keyword, we decided to use "i_f" instead.

```
#ifndef COMPLEX_SUFFIX_H_
#define COMPLEX_SUFFIX_H_
#include <complex>
namespace std{
inline namespace literals{
inline namespace complex_literals{
constexpr
std::complex<long double> operator"" il(long double d){
        return std::complex<long double>{0.0L,static_cast<long double>(d)};
}
constexpr
std::complex<long double> operator"" il(unsigned long long d){
        return std::complex<long double>{0.0L,static_cast<long double>(d)};
}
constexpr
std::complex<double> operator"" i(long double d){
        return std::complex<double>{0,static_cast<double>(d)};
}
constexpr
```

```
std::complex<double> operator"" i(unsigned long long d){
        return std::complex<double>{0,static_cast<double>(d)};
}
constexpr
std::complex<float> operator"" i_f(long double d){
        return std::complex<float>{0,static_cast<float>(d)};
}
constexpr
std::complex<float> operator"" i_f(unsigned long long d){
        return std::complex<float>{0,static_cast<float>(d)};
}
}}}
#endif /* COMPLEX_SUFFIX_H_ */
```

4.6 duration

Except for inline namespaces this hasn't been changed from the original version in N3402. And it includes the fix of the missing static member definition of "value".

```
#ifndef CHRONO_SUFFIX_H_
#define CHRONO_SUFFIX_H_
#include <chrono>
#include <limits>
#include "suffixes_parse_integers.h"
namespace std {
inline namespace literals {
inline namespace chrono_literals{

namespace __impl {
using namespace std::parse_int;

template <unsigned long long val, typename DUR>
struct select_type:conditional<
    (val <=static_cast<unsigned long long>(
            std::numeric_limits<typename DUR::rep>::max()))
        , DUR , void > {
                static constexpr typename select_type::type
                        value{ static_cast<typename select_type::type>(val) };
};
template <unsigned long long val, typename DUR>
constexpr typename select_type<val,DUR>::type select_type<val,DUR>::value;
}

template <char... Digits>
constexpr typename
__impl::select_type<__impl::base_dispatch<Digits...>::value,
std::chrono::hours>::type
operator"" h(){
```

```
        return  __impl::select_type<__impl::base_dispatch<Digits...>::value,
                std::chrono::hours>::value;
}
constexpr std::chrono::duration<long double, ratio<3600,1>>
operator"" h(long double hours){
        return std::chrono::duration<long double,ratio<3600,1>>{hours};
}
template <char... Digits>
constexpr typename
__impl::select_type<__impl::base_dispatch<Digits...>::value,
std::chrono::minutes>::type
operator"" min(){
        return __impl::select_type<__impl::base_dispatch<Digits...>::value,
               std::chrono::minutes>::value;
}
constexpr std::chrono::duration<long double, ratio<60,1>>
operator"" min(long double min){
        return std::chrono::duration<long double,ratio<60,1>>{min};
}

template <char... Digits>
constexpr typename
__impl::select_type<__impl::base_dispatch<Digits...>::value,
std::chrono::seconds>::type
operator"" s(){
        return __impl::select_type<__impl::base_dispatch<Digits...>::value,
               std::chrono::seconds>::value;
}
constexpr std::chrono::duration<long double>
operator"" s(long double sec){
        return std::chrono::duration<long double>{sec};
}

template <char... Digits>
constexpr typename
__impl::select_type<__impl::base_dispatch<Digits...>::value,
std::chrono::milliseconds>::type
operator"" ms(){
        return __impl::select_type<__impl::base_dispatch<Digits...>::value,
               std::chrono::milliseconds>::value;
}
constexpr std::chrono::duration<long double, milli>
operator"" ms(long double msec){
        return std::chrono::duration<long double,milli>{msec};
}

template <char... Digits>
constexpr typename
__impl::select_type<__impl::base_dispatch<Digits...>::value,
```

```
std::chrono::microseconds>::type
operator"" us(){
        return __impl::select_type<__impl::base_dispatch<Digits...>::value,
               std::chrono::microseconds>::value;
}
constexpr std::chrono::duration<long double, micro>
operator"" us(long double usec){
        return std::chrono::duration<long double, micro>{usec};
}

template <char... Digits>
constexpr typename
__impl::select_type<__impl::base_dispatch<Digits...>::value,
std::chrono::nanoseconds>::type
operator"" ns(){
        return __impl::select_type<__impl::base_dispatch<Digits...>::value,
               std::chrono::nanoseconds>::value;
}
constexpr std::chrono::duration<long double, nano>
operator"" ns(long double nsec){
        return std::chrono::duration<long double, nano>{nsec};
}

}}}
#endif /* CHRONO_SUFFIX_H_ */
```

Here are some test cases for the UDL operators to show how they can be applied/tested. Most just test compilability.

```
using namespace std::chrono_literals;

void testChronoLiterals(){
        static_assert(std::is_same<std::chrono::hours::rep, int>::value,
                      "hours are too long to check");
        //constexpr auto overflowh= 0x80000000h; // compile error!
        constexpr auto xh=5h;
        ASSERT_EQUAL(std::chrono::hours{5}.count(),xh.count());
        static_assert(std::chrono::hours{5}==xh,"chrono suffix hours");
        constexpr auto xmin=0x5min;
        static_assert(std::chrono::duration<unsigned long long,
                      std::ratio<60,1>>{5}==xmin,"chrono suffix min");
        constexpr auto x=05s;
        static_assert(std::chrono::duration<unsigned long long,
                      std::ratio<1,1>>{5}==x,"chrono suffix s");
        constexpr auto xms=5ms;
        static_assert(std::chrono::duration<unsigned long long,
                      std::ratio<1,1000>>{5}==xms,"chrono suffix ms");
        constexpr auto xus=5us;
        static_assert(std::chrono::duration<unsigned long long,
```

```
                        std::ratio<1,1000000>>{5}==xus,"chrono suffix ms");
        constexpr auto xns=5ns;
        static_assert(std::chrono::duration<unsigned long long,
                        std::ratio<1,1000000000>>{5}==xns,"chrono suffix ms");

        constexpr auto dh=0.5h;
        constexpr auto dmin=0.5min;
        constexpr auto ds=0.5s;
        constexpr auto dms=0.5ms;
        constexpr auto dus=0.5us;
        constexpr auto dns=0.5ns;
}
void aTestForDuration(){
        auto  x=5h;
        auto y=18000s;
        ASSERT_EQUAL(x,y);
}
```