# User-defined Literals for Standard Library Types

Peter Sommerlad

2012-08-31

| Document Number: | N3402=12-0092 |
|---|---|
| Date: | 2012-08-31 |
| Project: | Programming Language C++ |

## 1   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, b for binary representation of integers and i for imaginary parts of complex numbers.

Discussion on the reflector in May 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 a namespace for a (group of related) UDL operator(s)

- use a namespace within `std` for all such UDL namespaces, `std::suffixes` was suggested

- 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.

- s was proposed for `std::string` but is also ISO standard for seconds and could be convenient for `std::chrono::duration` values.

- an UDL for constructing `std::string` literals should not allocate memory, but use a `str_ref` type, once some like that is available in the standard.

- any proposal that is made for adding user-defined literal functions to the standard library will evoke some discussion.

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

- `unsigned` integers, suffix `b` plus further suffixes denoting resulting types as for integral constants in namespace `std::suffixes::binary`

- `std::basic_string`, suffix `s` in namespace `std::suffixes::string`

- `std::complex`, suffixes `i`, `li`, `fi`, `r`, `lr`, `fr` in namespace `std::suffixes::complex`

- `std::chrono::duration`, suffixes `h`, `min`, `s`, `ms`, `us`, `ns` in namespace `std::suffixes::chrono`

## 1.1 Open Issues

### 1.1.1 Upper-case versions of suffixes

While it seems useful and symmetric to provide upper case variations of suffixes `u`, `l`, `ll`, `ull` as allowed for integral constants, it needs to be discussed if also `'b'` should vary in case accordingly and thus doubling the number of overloaded UDL operators.

Similar discussions might be needed for complex numbers suffixes.

I have the opinion we should stick for lower case only for strings and chrono suffixes.

### 1.1.2 Suffix r for real-part only std::complex numbers

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.

## 1.2 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 feedback on early drafts and to Jonathan Wakely for guidelines on GCC command line options. Thanks to Alberto Ganesh Barbati for feedback on duration representation overflow.

## 2 Possible Implementation

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

## 2.1 binary

```
#ifndef BINARY_H_
#define BINARY_H_
#include <limits>
#include <type_traits>
namespace std{
namespace suffixes{
namespace binary{
namespace __impl{
// serves also as base case of recursion
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));
};

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

template <unsigned long long val, typename INTTYPE, typename... INTS>
struct select_type<val,INTTYPE,INTS...>:conditional<
        val<=static_cast<unsigned long long>(std::numeric_limits<INTTYPE>::max())
        ,INTTYPE
        ,typename select_type<val,INTS...>::type >{
        static typename select_type::type const
                value{ static_cast<typename select_type::type>(val) };
};

template <unsigned long long val>
struct select_type<val>{
        typedef unsigned long long type;
        static type const value{  val };
};

}
```

3

```
template <char... Digits>
constexpr typename
__impl::select_type<__impl::bitsImpl<Digits...>::value,
      int, unsigned, long, unsigned long, long long>::type
operator"" b(){
        return  __impl::select_type<__impl::bitsImpl<Digits...>::value,
                        int, unsigned, long, unsigned long, long long>::value;
}
template <char... Digits>
constexpr typename
__impl::select_type<__impl::bitsImpl<Digits...>::value,
      long, unsigned long, long long>::type
operator"" bl(){
        return  __impl::select_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_type<__impl::bitsImpl<Digits...>::value,
      long long>::type
operator"" bll(){
        return  __impl::select_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_type<__impl::bitsImpl<Digits...>::value,
      unsigned, unsigned long>::type
operator"" bu(){
        return  __impl::select_type<__impl::bitsImpl<Digits...>::value,
                        unsigned, unsigned long>::value;
}

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

```cpp
        return  operator "" bu<Digits...>();
}

template <char... Digits>
constexpr typename
__impl::select_type<__impl::bitsImpl<Digits...>::value,
       unsigned long>::type
operator"" bul(){
        return  __impl::select_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
```

5

```
} //suffixes
} // std



#endif /* BINARY_H_ */
```

## 2.2   basic_string

```
#ifndef STRING_SUFFIX_H_
#define STRING_SUFFIX_H_
#include <string>
namespace std{
namespace suffixes{
namespace string{
#if 0 // less typing variant
#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_ */
```

## 2.3   std::complex

```
namespace std{
namespace suffixes{
namespace complex{
constexpr
std::complex<long double> operator"" _li(long double d){
        return std::complex<long double>{0,d};
}
constexpr
std::complex<long double> operator"" _li(unsigned long long d){
        return std::complex<long double>{0,static_cast<long double>(d)};
}
constexpr
std::complex<long double> operator"" _lr(long double d){
        return std::complex<long double>{d,0};
}
constexpr
std::complex<long double> operator"" _lr(unsigned long long d){
        return std::complex<long double>{static_cast<long double>(d),0};
}
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<double> operator"" _r(long double d){
        return std::complex<double>{static_cast<double>(d),0};
}
constexpr
std::complex<double> operator"" _r(unsigned long long d){
        return std::complex<double>{static_cast<double>(d),0};
}
constexpr
std::complex<float> operator"" _fi(long double d){
        return std::complex<float>{0,static_cast<float>(d)};
}
constexpr
std::complex<float> operator"" _fi(unsigned long long d){
        return std::complex<float>{0,static_cast<float>(d)};
}
constexpr
```

```
std::complex<float> operator"" _fr(long double d){
        return std::complex<float>{static_cast<float>(d),0};
}
constexpr
std::complex<float> operator"" _fr(unsigned long long d){
        return std::complex<float>{static_cast<float>(d),0};
}
}
}
}
```

## 2.4   duration

```
#ifndef CHRONO_SUFFIX_H_
#define CHRONO_SUFFIX_H_
#include <chrono>
#include <limits>
namespace std {
namespace suffixes {
namespace chrono {

namespace __impl {

template <char... Digits>
struct digImpl{
        static_assert(! sizeof...(Digits), "invalid decimal constant");
        static constexpr unsigned long long value=0;
};
template <char... Digits>
struct octImpl{
        static_assert(! sizeof...(Digits), "invalid octal constant");
        static constexpr unsigned long long value=0;
};
template <char... Digits>
struct hexImpl{
        static_assert(! sizeof...(Digits), "invalid hexadecimal constant");
        static constexpr unsigned long long value=0;
};
template <char... Digits>
struct digDispatch;

template <char... Digits>
struct digDispatch<'0','x',Digits...>{
        static constexpr unsigned long long value=hexImpl<Digits...>::value;
};
template <char... Digits>
struct digDispatch<'0',Digits...>{
        static constexpr unsigned long long value=octImpl<Digits...>::value;
};
```

```
template <char... Digits>
struct digDispatch{
        static constexpr unsigned long long value=digImpl<Digits...>::value;
};


template <size_t power>
struct ten_to{
static constexpr unsigned long long     val{ 10 * ten_to<power-1>::val};
};
template <>
struct ten_to<0>{
        static constexpr unsigned long long     val{ 1 };

};
template <size_t power>
struct eight_to{
static constexpr unsigned long long     val{ 8 * eight_to<power-1>::val};
};
template <>
struct eight_to<0>{
        static constexpr unsigned long long     val{ 1 };

};
template <size_t power>
struct hex_to{
static constexpr unsigned long long     val{ 16 * hex_to<power-1>::val};
};
template <>
struct hex_to<0>{
        static constexpr unsigned long long     val{ 1 };

};
template <char... Digits>
struct octImpl<'0',Digits...>{
        static constexpr unsigned long long value{ octImpl<Digits...>::value};
};
template <char... Digits>
struct octImpl<'1',Digits...>{
        static constexpr unsigned long long value{ 1 *eight_to<sizeof...(Digits)>::val
                                        + octImpl<Digits...>::value};
};
template <char... Digits>
struct octImpl<'2',Digits...>{
        static constexpr unsigned long long value{ 2 *eight_to<sizeof...(Digits)>::val
                                        + octImpl<Digits...>::value};
};
template <char... Digits>
struct octImpl<'3',Digits...>{
```

```cpp
        static constexpr unsigned long long value{ 3 *eight_to<sizeof...(Digits)>::val
                                                + octImpl<Digits...>::value};
};
template <char... Digits>
struct octImpl<'4',Digits...>{
        static constexpr unsigned long long value{ 4 *eight_to<sizeof...(Digits)>::val
                                                + octImpl<Digits...>::value};
};
template <char... Digits>
struct octImpl<'5',Digits...>{
        static constexpr unsigned long long value{ 5 *eight_to<sizeof...(Digits)>::val
                                                + octImpl<Digits...>::value};
};
template <char... Digits>
struct octImpl<'6',Digits...>{
        static constexpr unsigned long long value{ 6 *eight_to<sizeof...(Digits)>::val
                                                + octImpl<Digits...>::value};
};
template <char... Digits>
struct octImpl<'7',Digits...>{
        static constexpr unsigned long long value{ 7 *eight_to<sizeof...(Digits)>::val
                                                + octImpl<Digits...>::value};
};


template <char... Digits>
struct digImpl<'0',Digits...>{
        static constexpr unsigned long long value{ digImpl<Digits...>::value};
};
template <char... Digits>
struct digImpl<'1',Digits...>{
        static constexpr unsigned long long value{ 1 *ten_to<sizeof...(Digits)>::val
                                                + digImpl<Digits...>::value};
};
template <char... Digits>
struct digImpl<'2',Digits...>{
        static constexpr unsigned long long value{ 2 *ten_to<sizeof...(Digits)>::val
                                                + digImpl<Digits...>::value};
};
template <char... Digits>
struct digImpl<'3',Digits...>{
        static constexpr unsigned long long value{ 3 *ten_to<sizeof...(Digits)>::val
                                                + digImpl<Digits...>::value};
};
template <char... Digits>
struct digImpl<'4',Digits...>{
        static constexpr unsigned long long value{ 4 *ten_to<sizeof...(Digits)>::val
                                                + digImpl<Digits...>::value};
};
template <char... Digits>
```

```cpp
struct digImpl<'5',Digits...>{
        static constexpr unsigned long long value{ 5 *ten_to<sizeof...(Digits)>::val
                                                  + digImpl<Digits...>::value};
};
template <char... Digits>
struct digImpl<'6',Digits...>{
        static constexpr unsigned long long value{ 6 *ten_to<sizeof...(Digits)>::val
                                                  + digImpl<Digits...>::value};
};
template <char... Digits>
struct digImpl<'7',Digits...>{
        static constexpr unsigned long long value{ 7 *ten_to<sizeof...(Digits)>::val
                                                  + digImpl<Digits...>::value};
};
template <char... Digits>
struct digImpl<'8',Digits...>{
        static constexpr unsigned long long value{ 8 *ten_to<sizeof...(Digits)>::val
                                                  + digImpl<Digits...>::value};
};
template <char... Digits>
struct digImpl<'9',Digits...>{
        static constexpr unsigned long long value{ 9 *ten_to<sizeof...(Digits)>::val
                                                  + digImpl<Digits...>::value};
};

template <char... Digits>
struct hexImpl<'0',Digits...>{
        static constexpr unsigned long long value{ hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'1',Digits...>{
        static constexpr unsigned long long value{ 1 *hex_to<sizeof...(Digits)>::val
                                                  + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'2',Digits...>{
        static constexpr unsigned long long value{ 2 *hex_to<sizeof...(Digits)>::val
                                                  + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'3',Digits...>{
        static constexpr unsigned long long value{ 3 *hex_to<sizeof...(Digits)>::val
                                                  + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'4',Digits...>{
        static constexpr unsigned long long value{ 4 *hex_to<sizeof...(Digits)>::val
                                                  + hexImpl<Digits...>::value};
};
```

```
template <char... Digits>
struct hexImpl<'5',Digits...>{
        static constexpr unsigned long long value{ 5 *hex_to<sizeof...(Digits)>::val
                                                + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'6',Digits...>{
        static constexpr unsigned long long value{ 6 *hex_to<sizeof...(Digits)>::val
                                                + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'7',Digits...>{
        static constexpr unsigned long long value{ 7 *hex_to<sizeof...(Digits)>::val
                                                + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'8',Digits...>{
        static constexpr unsigned long long value{ 8 *hex_to<sizeof...(Digits)>::val
                                                + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'9',Digits...>{
        static constexpr unsigned long long value{ 9 *hex_to<sizeof...(Digits)>::val
                                                + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'a',Digits...>{
        static constexpr unsigned long long value{ 0xa *hex_to<sizeof...(Digits)>::val
                                                + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'b',Digits...>{
        static constexpr unsigned long long value{ 0xb *hex_to<sizeof...(Digits)>::val
                                                + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'c',Digits...>{
        static constexpr unsigned long long value{ 0xc *hex_to<sizeof...(Digits)>::val
                                                + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'d',Digits...>{
        static constexpr unsigned long long value{ 0xd *hex_to<sizeof...(Digits)>::val
                                                + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'e',Digits...>{
        static constexpr unsigned long long value{ 0xe *hex_to<sizeof...(Digits)>::val
                                                + hexImpl<Digits...>::value};
```

```
};
template <char... Digits>
struct hexImpl<'f',Digits...>{
        static constexpr unsigned long long value{ 0xf *hex_to<sizeof...(Digits)>::val
                                                  + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'A',Digits...>{
        static constexpr unsigned long long value{ 0xa *hex_to<sizeof...(Digits)>::val
                                                  + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'B',Digits...>{
        static constexpr unsigned long long value{ 0xb *hex_to<sizeof...(Digits)>::val
                                                  + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'C',Digits...>{
        static constexpr unsigned long long value{ 0xc *hex_to<sizeof...(Digits)>::val
                                                  + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'D',Digits...>{
        static constexpr unsigned long long value{ 0xd *hex_to<sizeof...(Digits)>::val
                                                  + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'E',Digits...>{
        static constexpr unsigned long long value{ 0xe *hex_to<sizeof...(Digits)>::val
                                                  + hexImpl<Digits...>::value};
};
template <char... Digits>
struct hexImpl<'F',Digits...>{
        static constexpr unsigned long long value{ 0xf *hex_to<sizeof...(Digits)>::val
                                                  + hexImpl<Digits...>::value};
};


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) };
        };
} // namespace __impl
```

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

template <char... Digits>
constexpr typename
__impl::select_type<__impl::digDispatch<Digits...>::value,std::chrono::milliseconds>::type
operator"" ms(){
        return __impl::select_type<
        __impl::digDispatch<Digits...>::value,std::chrono::milliseconds>::value;
}
constexpr std::chrono::duration<long double, ratio<1,1000>> operator"" ms(long double msec)
        return std::chrono::duration<long double,ratio<1,1000>>{msec};
}

template <char... Digits>
constexpr typename
__impl::select_type<__impl::digDispatch<Digits...>::value,std::chrono::microseconds>::type
operator"" us(){
        return __impl::select_type<
        __impl::digDispatch<Digits...>::value,std::chrono::microseconds>::value;
```

```
}
constexpr std::chrono::duration<long double, ratio<1,1000000>> operator"" us(long double usec){
        return std::chrono::duration<long double,ratio<1,1000000>>{usec};
}

template <char... Digits>
constexpr typename
__impl::select_type<__impl::digDispatch<Digits...>::value,std::chrono::nanoseconds>::type
operator"" ns(){
        return __impl::select_type<
        __impl::digDispatch<Digits...>::value,std::chrono::nanoseconds>::value;
}
constexpr std::chrono::duration<long double, ratio<1,1000000000>> operator"" ns(long double nsec){
        return std::chrono::duration<long double,ratio<1,1000000000>>{nsec};
}
}}}
#endif /* CHRONO_SUFFIX_H_ */
```

# 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 suffixes for collecting standard UDLs

As a common schema this paper proposes to put all suffixes for user defined literals in separate namespaces that are below the namespace `std::suffixes`.

## 3.2 operator"" b() etc. for binary integer literals

Append a subclause [suffixes.binary] to clause [utilities] and expand the table in [utilities.general] accordingly.

Insert the subclause [suffixes.binary]

## 3.3 Binary integer literals                    [suffixes.binary]

[1] This subclause contains user-defined literal operators for representing binary encoded integer literals.

**Header `<suffix_binary>` synopsis**

```
namespace std{
namespace suffixes{
namespace binary{

template <char... Digits>
```

```
    constexpr see below
    operator"" b();
    template <char... Digits>
    constexpr see below
    operator"" bu();
    template <char... Digits>
    constexpr see below
    operator"" bU();
    template <char... Digits>
    constexpr see below
    operator"" bl();
    template <char... Digits>
    constexpr see below
    operator"" bL();
    template <char... Digits>
    constexpr see below
    operator"" bul();
    template <char... Digits>
    constexpr see below
    operator"" buL();
    template <char... Digits>
    constexpr see below
    operator"" bll();
    template <char... Digits>
    constexpr see below
    operator"" bLL();
    template <char... Digits>
    constexpr unsigned long long
    operator"" bull();
    template <char... Digits>
    constexpr unsigned long long
    operator"" buLL();
    template <char... Digits>
    constexpr unsigned long long
    operator"" bUll();
    template <char... Digits>
    constexpr unsigned long long
    operator"" bULL();
    } // binary
    } //suffixes
    } // std
```

[2]   A binary integer literal is a sequence of the binary digits '0' (zero) or '1' (one) that
is followed by one of the suffixes in namespace `std::suffixes::binary`. If there is any
other digit in a binary integer literal the program is ill-formed. The lexically first digit
of the sequence of digits is the most significant. The sequence of binary digits forming a
binary literal create an integral value that corresponds to its interpretation as a binary
number.

[3]   The type of the binary literal is determined from the its value and the additional suffix

(`u`, `l`, `ul`, `ull` and their uppercase variants as with other integer literals) to `b` like the determination of the type of octal integer literals in clause [lex.icon] p 2.

4  [*Example:* The following code shows some binary literals. The type of xll is adjusted due to its large value, assuming `sizeof(long) < 8` and `char` as octet.

```
{
    using namespace std::suffixes::binary;
    auto x=100b;  // int x=4
    auto y=1000bu;  // unsigned y=8
    auto xll=011111111111111111111111111111111111111111111111111111111111111b;
        // long long xll
}
```

— *end example*]

```
template <char... Digits>
constexpr see below
operator"" b();
```

5      *Effects:* Creates an integral constant with the value determined as described above. The return type is determined according to the first row of table 6 in clause [lex.icon] p 2 and the column for octal literals.

```
template <char... Digits>
constexpr see below
operator"" bu();
template <char... Digits>
constexpr see below
operator"" bU();
```

6      *Effects:* Creates an integral value determined as described above. The return type is determined according to the second row of table 6 in clause [lex.icon] p 2 and the column for octal literals.

```
template <char... Digits>
constexpr see below
operator"" bl();
template <char... Digits>
constexpr see below
operator"" bL();
```

7      *Effects:* Creates an integral value determined as described above. The return type is determined according to the third row of table 6 in clause [lex.icon] p 2 and the column for octal literals.

```
template <char... Digits>
constexpr see below
operator"" bul();
```

```
template <char... Digits>
constexpr see below
operator"" buL();
template <char... Digits>
constexpr see below
operator"" bUl();
template <char... Digits>
constexpr see below
operator"" bUL();
```

8    *Effects:* Creates an integral value determined as described above. The return type
     is determined according to the fourth row of table 6 in clause [lex.icon] p 2 and
     the column for octal literals.

```
template <char... Digits>
constexpr see below
operator"" bll();
template <char... Digits>
constexpr see below
operator"" bLL();
```

9    *Effects:* Creates an integral constant with the value determined as described above.
     The return type is determined according to the fifth row of table 6 in clause
     [lex.icon] p 2 and the column for octal literals.

```
template <char... Digits>
constexpr unsigned long long
operator"" bull();
template <char... Digits>
constexpr unsigned long long
operator"" buLL();
template <char... Digits>
constexpr unsigned long long
operator"" bUll();
template <char... Digits>
constexpr unsigned long long
operator"" bULL();
```

10   *Effects:* Creates an integral value determined as described above.

## 3.4   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 namespace std::suffixes::string

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

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

## 3.5  Suffix for basic_string literals      [basic.string.suffixes]

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

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

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

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

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

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

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

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

## 3.6  UDL operators for std::complex

Make the following additions and changes to library subclause 26.4 [complex.numbers] to accommodate user-defined literal suffixes for complex number literals.

Insert in subclause 26.4.1 [complex.syn] in the synopsis at the appropriate place the namespace std::suffixes::complex

```
namespace std{
namespace suffixes{
namespace complex{
constexpr std::complex<long double> operator"" li(long double);
constexpr std::complex<long double> operator"" LI(long double);
constexpr std::complex<long double> operator"" li(unsigned long long);
constexpr std::complex<long double> operator"" LI(unsigned long long);
```

```
        constexpr std::complex<long double> operator"" lr(long double);
        constexpr std::complex<long double> operator"" LR(long double);
        constexpr std::complex<long double> operator"" lr(unsigned long long);
        constexpr std::complex<long double> operator"" LR(unsigned long long);
        constexpr std::complex<double> operator"" i(long double);
        constexpr std::complex<double> operator"" I(long double);
        constexpr std::complex<double> operator"" i(unsigned long long);
        constexpr std::complex<double> operator"" I(unsigned long long);
        constexpr std::complex<double> operator"" r(long double);
        constexpr std::complex<double> operator"" R(long double);
        constexpr std::complex<double> operator"" r(unsigned long long);
        constexpr std::complex<double> operator"" R(unsigned long long);
        constexpr std::complex<float> operator"" fi(long double);
        constexpr std::complex<float> operator"" FI(long double);
        constexpr std::complex<float> operator"" fi(unsigned long long);
        constexpr std::complex<float> operator"" FI(unsigned long long);
        constexpr std::complex<float> operator"" fr(long double);
        constexpr std::complex<float> operator"" FR(long double);
        constexpr std::complex<float> operator"" fr(unsigned long long);}
        constexpr std::complex<float> operator"" FR(unsigned long long);}
        }
        }
```

Append a new subclause after subclause 26.4.10 [ccmplx] as follows

## 3.7  Suffix for complex number literals    [complex.suffixes]

[1]  This section describes literal suffixes for constructing complex number literals.  The suffixes i, li, fi create complex numbers with their imaginary part denoted by the given literal number and the real part being zero of the types `complex<double>`, `complex<long double>`, and `complex<float>` respectively.

The suffixes r, lr, fr create complex numbers with the real part denoted by the given literal number and the imaginary part being zero of the types `complex<double>`, `complex<long double>`, and `complex<float>` respectively.

```
constexpr std::complex<long double> operator"" li(long double d);
constexpr std::complex<long double> operator"" LI(long double d);
constexpr std::complex<long double> operator"" li(unsigned long long d);
constexpr std::complex<long double> operator"" LI(unsigned long long d);
```

[2]    *Effects:* Creates a complex literal as `std::complex<long double>{0.0L, static_-cast<long double>(d)}`.

```
constexpr std::complex<double> operator"" i(long double d);
constexpr std::complex<double> operator"" I(long double d);
constexpr std::complex<double> operator"" i(unsigned long long d);
constexpr std::complex<double> operator"" I(unsigned long long d);
```

3      *Effects:* Creates a complex literal as `std::complex<double>{0.0, static_cast<double>(d)}`.

```
constexpr std::complex<float> operator"" fi(long double d);
constexpr std::complex<float> operator"" FI(long double d);
constexpr std::complex<float> operator"" fi(unsigned long long d);
constexpr std::complex<float> operator"" FI(unsigned long long d);
```

4      *Effects:* Creates a complex literal as `std::complex<float>{0.0f, static_cast<float>(d)}`.

```
constexpr std::complex<long double> operator"" lr(long double d);
constexpr std::complex<long double> operator"" LR(long double d);
constexpr std::complex<long double> operator"" lr(unsigned long long d);
constexpr std::complex<long double> operator"" LR(unsigned long long d);
```

5      *Effects:* Creates a complex literal as `std::complex<long double>{static_cast<long double>(d), 0.0L}`.

```
constexpr std::complex<double> operator"" r(long double d);
constexpr std::complex<double> operator"" R(long double d);
constexpr std::complex<double> operator"" r(unsigned long long d);
constexpr std::complex<double> operator"" R(unsigned long long d);
```

6      *Effects:* Creates a complex literal as `std::complex<double>{static_cast<double>(d), 0.0}`.

```
constexpr std::complex<float> operator"" fr(long double d);
constexpr std::complex<float> operator"" FR(long double d);
constexpr std::complex<float> operator"" fr(unsigned long long d);}
constexpr std::complex<float> operator"" FR(unsigned long long d);}
```

7      *Effects:* Creates a complex literal as `std::complex<float>{static_cast<float>(d), 0.0f}`.

## 3.8 Suffixes for std::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 namespace std::suffixes::chrono

```
namespace std {
namespace suffixes {
namespace chrono {
constexpr
std::chrono::hours operator"" h(unsigned long long);
```

```
constexpr
std::chrono::duration<see below, ratio<3600,1>> operator"" h(long double);
constexpr
std::chrono::minutes operator"" min(unsigned long long);
constexpr
std::chrono::duration<see below, ratio<60,1>> operator"" min(long double);
constexpr
std::chrono::seconds operator"" s(unsigned long long);
constexpr
std::chrono::duration<see below, ratio<1,1>> operator"" s(long double);
constexpr
std::chrono::milliseconds operator"" ms(unsigned long long);
constexpr
std::chrono::duration<see below, ratio<1,1000>> operator"" ms(long double);
constexpr
std::chrono::microseconds operator"" us(unsigned long long);
constexpr
std::chrono::duration<see below, ratio<1,1000000>> operator"" us(long double);
constexpr
std::chrono::nanoseconds operator"" ns(unsigned long long);
constexpr
std::chrono::duration<see below, ratio<1,1000000000>> 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.suffixes] as follows.

### 3.8.1   Suffix for duration literals            [time.duration.suffixes]

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 the above suffixes are applied to a floating point literal the result is a std::duration literal with an implementation-defined floating point representation.

3   If the above 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::suffixes::chrono;
    auto constexpr aday=24h;
    auto constexpr lesson=45min;
    auto constexpr halfanhour=0.5h;
}
```

*— end example* ]

5    [ *Note:* The suffix for microseconds is `us`, but if unicode identifiers are allowed implementations are encouraged to provide $\mu$`s` as well.  *— end note* ]

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

6      *Effects:* Creates a `duration` literal representing `hours` hours.

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

7      *Effects:* Creates a `duration` literal representing `min` minutes.

```
constexpr
std::chrono::seconds operator"" s(unsigned long long sec);
constexpr
std::chrono::duration<see below, ratio<1,1>> operator"" s(long double sec);
```

8      *Effects:* Creates a `duration` literal representing `sec` seconds.

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

```
constexpr
std::chrono::milliseconds operator"" ms(unsigned long long msec);
constexpr
std::chrono::duration<see below, ratio<1,1000>> operator"" ms(long double msec);
```

9      *Effects:* Creates a `duration` literal representing `msec` milliseconds.

```
constexpr
std::chrono::microseconds operator"" us(unsigned long long usec);
constexpr
std::chrono::duration<see below, ratio<1,1000000>> operator"" us(long double usec);
```

10      *Effects:* Creates a `duration` literal representing `usec` microseconds.

```
constexpr
std::chrono::nanoseconds operator"" ns(unsigned long long nsec);
constexpr
std::chrono::duration<see below, ratio<1,1000000000>> operator"" ns(long double nsec);
```

11    *Effects:* Creates a `duration` literal representing `nsec` nanoseconds.