User-defined Literals for Standard Library Types

Peter Sommerlad

2012-09-08

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.
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.

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 Rationale

User-defined literal operators (UDL) are a new feature 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 binary number, imaginary number, or `std::string` literals.

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

In addition to a facility for binary literals for integral values, 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).

During the creation of this paper, it became apparent that a mechanism for parsing integral values from UDL operator template can be useful in its own, so that implementers of their own UDL operators can reuse it. A further observation from implementing binary literals was, that to mimic the mechanism of the compiler to automatically select a best fitting integral type, based on the literal’s value can be reused as well in similar UDL operators. The last mechanism actually requires parsing integral values by UDL operator templates to retain the value as a compile-time constant. This would get lost if the UDL operator taking an `unsigned long long` would have been chosen.

### 1.2 Open Issues

#### 1.2.1 Suffixes Utilities

It has to be decided if the utilities for implementing UDL suffix operators with integers should be standardized.
The template `select_int_type` might be a candidate for the clause [meta.type.synop], aka header `<type_traits>`.

1.2.2 Upper-case versions of suffixes

While it seems useful and symmetric to provide upper case variations of suffixes u, 1, 11, 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.

1.2.3 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.

1.3 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 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.

2 Possible Implementation

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

2.1 integer parsing

For its usage, see the implementation of std::chrono::duration literals.

```cpp
#ifndef SUFFIXESPARENUMBERS_H_
#define SUFFIXESPARENUMBERS_H_
#include <cstddef>
namespace std {
    namespace suffixes {
        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};

4
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};
};


```cpp
};

template <unsigned base, char... Digits>
struct parse_int<base, 'f', 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;
};
```
2.2 integral type fitting

For its usage, see the implementation of binary literals.

```cpp
#ifndef SELECT_INT_TYPE_H_
#define SELECT_INT_TYPE_H_
#include <type_traits>
#include <limits>
namespace std {
namespace suffixes {
namespace select_int_type {

    template <unsigned long long val, typename... INTS>
    struct select_int_type {
        typedef unsigned long long type;
        static type const value{ val };
    }

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

    template <unsigned long long val>
    struct select_int_type<val>{
        typedef unsigned long long type;
        static type const value{ val };
    };
}
}
#endif /* SUFFIXESPARSENUMBERS.H */
```
2.3 binary

```cpp
#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::suffixes::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>::type
b(){
    return __impl::select_int_type<__impl::bitsImpl<Digits...>::value,
        int, unsigned, long, unsigned long, long long>::value;
}
```
long, unsigned long, long long>::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>::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>::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>::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 */

2.4 basic_string

#ifndef STRING_SUFFIX_H_
define STRING_SUFFIX_H_
#include <string>
namespace std{
namespace suffixes{
namespace string{
2.5  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)};
    }
}  
}  
}  
}/* STRING_SUFFIX_H_ */
2.6 duration

 ifndef CHRONO_SUFFIX_H_
 #define CHRONO_SUFFIX_H_
```cpp
#include <chrono>
#include <limits>
#include "suffixes_parse_integers.h"
namespace std {
 namespace suffixes {
 namespace chrono {
 namespace __impl {
 using namespace std::suffixes::parse_int;

template <unsigned long long val, typename DUR>
 struct select_type:
 condition<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 <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"m"(){
 return __impl::select_type<__impl::base_dispatch<Digits...>::value,
 std::chrono::minutes>::value;
 }

constexpr std::chrono::duration<long double, ratio<60,1>> operator"m"(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, 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::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, 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::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, 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::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, ratio<1,1000000000>>
operator"ns(long double nsec){
    return std::chrono::duration<long double, ratio<1,1000000000>>{nsec};
}}}}

```
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 Suffixes Integer Parsing Utilities

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

3.3 Parsing integer literals [suffixes.parseint]

1 This subclause contains helper template classes for implementing template user-defined literal operators for parsing integer literals.

   Header <suffix_parse> synopsis
   
   namespace std {
   namespace suffixes {

   template <char... Digits>
   struct base_dispatch;

   template <unsigned base, char... Digits>
   struct parse_int;

   } // suffixes
   } // std

2 The class templates base_dispatch and parse_int are intended to be used by implementors of a user-defined-literal operator template for parsing integral values. They both provide a static constexpr unsigned long long member variable named value.

3 Class template parse_int can be used when the numerical base is known and not to be determined from Digits....

4 The class template base_dispatch will determine the numerical base to be used like the compiler does with integral literals, i.e., a number starting with 0 (zero) will be considered octal, a number starting with 0x or 0X will be considered hexadecimal and all other numbers will be considered decimal.

5 [Example:

   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
— end example ]

template <char... Digits>
struct base_dispatch;

Effects: Creates an integral constant as its static constexpr member variable value which is of type unsigned long long with the value determined from the first or first two char values in Digits... as described below.

If Digits... starts with '0', 'x' or '0', 'X' the remaining characters are parsed as a hexadecimal number. If there are no such subsequent characters or Digits... contains characters beyond those for hexadecimal literals the program is ill formed.

If Digits... starts with '0' the remaining characters are parsed as an octal number. If there are characters in Digits... that are not octal digits ('0'- '7') the program is ill-formed.

Otherwise Digits... is interpreted as a decimal number. If there are characters in Digits... that are not decimal digits, the program is ill formed.

template <unsigned base, char... Digits>
struct parse_int;

Requires: base > 1 && base <= 16

Effects: Creates an integral constant as its static constexpr member variable value which is of type unsigned long long with the value determined by parsing Digits... as an integer literal of the given base.

If Digits... contains characters outside the digits of the given base, the program is ill-formed.

[Example:

    template <char... Digits>
    constexpr unsigned long long
    operator"" _ternary(){
        return std::suffixes::parse_int<3,Digits...>::value;
    }
    constexpr auto three= 010_ternary;
    static_assert(three==3, "_ternary should be three-based");
    constexpr auto invalid=3_ternary; // ill-formed.
]
3.4 Integer Constant Type Selection

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

3.5 Select Matching Type for Integer Constants
[suffixes.selectinttype]

This subclause contains helper template classes for performing the determination of the type of integer literals as in clause [lex.icon] p 2.

Header <select_int_type> synopsis

namespace std {

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

Example: The class template will be used with a list of integral types as those given in clause [lex.icon] p 2 in one of the table 6. The following code demonstrates its use.

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

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

Requires: INTS... must consist of a list of integral types Ti where numeric_limits<Ti>::max() is less or equal of numeric_limits<Ti+1>::max().
Effects: The member \texttt{type} of \texttt{select\_int\_type} corresponds to the first type \texttt{T} in \texttt{INTS...} where \texttt{val} \leq \texttt{static\_cast<\texttt{unsigned\_long\_long}\>(numeric\_limits<\texttt{T}>::max())}. If no such type exists the member \texttt{type} is \texttt{unsigned\_long\_long}.

The member \texttt{value} is of type \texttt{select\_int\_type::type} and its value is \texttt{val}.

\textit{Note:} No overflow can occur unless \texttt{val} already was produced by an overflowing operation. — end note

3.6 \texttt{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]

3.7 Binary integer literals \texttt{[suffixes.binary]}

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

\texttt{Header <suffix\_binary> synopsis}

\begin{verbatim}
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"" bl();
      template <char... Digits>
      constexpr see below
      operator"" buL();
    }
  }
}
\end{verbatim}
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.

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.

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.

```cpp
{
    using namespace std::suffixes::binary;
    auto x=100b; // int x=4
    auto y=1000bu; // unsigned y=8
    auto xll=0111111111111111111111111111111111111111111111111111111111111111b;
    // long long xll
}
```

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();

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();

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();

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.
**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.

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

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

### 3.8 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

```cpp
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]

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

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

*Returns:* basic_string<char>[str,len]

```cpp
basic_string<wchar_t> operator "" s(wchar_t const *str, size_t len);
```
Returns: basic_string<wchar_t>{str,len}

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

Returns: basic_string<char16_t>{str,len}

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

Returns: basic_string<char32_t>{str,len}

3.10 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);
}}}

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

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

```cpp
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<double>{0.0, static_cast<double>(d)}.

```cpp
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<float>{0.0f, static_cast<float>(d)}.

```cpp
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<long double>{static_cast<long double>(d), 0.0L}.

```cpp
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}.

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

```cpp
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);
```

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

### 3.12 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`

```cpp
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.

3.12.1 Suffix for duration literals

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, milliseconds, microseconds, and nanoseconds respectively if they are applied to integral literals.

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.

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.

Example: The following code shows some duration literals.

```cpp
{  
    using namespace std::suffixes::chrono;
    auto constexpr aday=24h;
    auto constexpr lesson=45min;
    auto constexpr halfanhour=0.5h;
}
```

Note: The suffix for microseconds is us, but if unicode identifiers are allowed implementations are encouraged to provide µs as well.

Effects: Creates a duration literal representing hours(hours).

Effects: Creates a duration literal representing minutes(min).

Effects: Creates a duration literal representing seconds(sec).
[Note: The same suffix \texttt{s} is used for \texttt{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]

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

\textbf{Effects:} Creates a \texttt{duration} literal representing \texttt{msec} milliseconds.

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

\textbf{Effects:} Creates a \texttt{duration} literal representing \texttt{usec} microseconds.

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

\textbf{Effects:} Creates a \texttt{duration} literal representing \texttt{nsec} nanoseconds.