

Doc no: N2349=07-0209
Date: 2007-07-19
Reply-To: Gabriel Dos Reis
gdr@cs.tamu.edu

Constant Expressions in the Standard Library — Revision 2

Gabriel Dos Reis

Bjarne Stroustrup

Texas A&M University

Abstract

This companion paper to the `constexpr` proposal suggests changes to the C++ Standard Library specification to take advantage of the generalized constant expression core language facility.

1 Clause 18: Language support library

Section §18.1. Modify paragraph §18.1/4 as follows:

The macro `offsetof(type, member-designator)` accepts a restricted set of *type* arguments in this International Standard. If *type* is not a ~~POD-structure or a POD-union (clause 9)~~ **literal type (3.9)**, the results are undefined.¹⁸⁴⁾ [...]

Section §18.2.1. Modify paragraph §18.2.1/3 as follows:

For all members declared `static const` **`constexpr`** in the `numeric_limits` template, specializations shall define these values in such a way that they are usable as ~~integral~~ constant expressions.

Section §18.2.1.1. In paragraph §18.2.1.1, redefine the class template `numeric_limits` as follows:

```
namespace std {
    template<class T> struct numeric_limits {
        static constexpr constexpr bool is_specialized = false;
        static constexpr T min() throw();
        static constexpr T max() throw();

        static constexpr constexpr int digits = 0;
        static constexpr constexpr int digits10 = 0;
        static constexpr constexpr int max_digits10 = 0;
        static constexpr constexpr bool is_signed = false;
        static constexpr constexpr bool is_integer = false;
        static constexpr constexpr bool is_exact = false;
        static constexpr constexpr int radix = 0;
        static constexpr T epsilon() throw();
        static constexpr T round_error() throw();

        static constexpr constexpr int min_exponent = 0;
        static constexpr constexpr int min_exponent10 = 0;
        static constexpr constexpr int max_exponent = 0;
        static constexpr constexpr int max_exponent10 = 0;

        static constexpr constexpr bool has_infinity = false;
        static constexpr constexpr bool has_quiet_NaN = false;
        static constexpr constexpr bool has_signaling_NaN = false;
        static constexpr constexpr float_denorm_style has_denorm
            = denorm_absent;
        static constexpr constexpr bool has_denorm_loss = false;
        static constexpr T infinity() throw();
        static constexpr T quiet_NaN() throw();
        static constexpr T signaling_NaN() throw();
        static constexpr T denorm_min() throw();

        static constexpr constexpr bool is_iec559 = false;
        static constexpr constexpr bool is_bounded = false;
        static constexpr constexpr bool is_modulo = false;

        static constexpr constexpr bool traps = false;
        static constexpr constexpr bool tinyness_before = false;
        static constexpr constexpr float_round_style round_style
            = round_toward_zero;
    };
};
```

```
};  
}
```

Section §18.2.1.2. Modify all of section §18.2.1.2 to propagate all the declaration changes above.

Section §18.2.1.5. Modify section §18.2.1.5 so that `numeric_limits<float>` and `numeric_limits<bool>` members are `constexpr`.

2 Clause 20: General utility library

Section §20.2. Ideally, relational operators and comparison operators on `pair<T, U>` should be `constexpr` when the corresponding operators for the template arguments `T` and `U` are `constexpr`. We postpone modifications to this part of the library until after the details of “concepts” are known.

Section §20.4. Much of the syntactic clutter of “type traits” could be simplified using `constexpr`. However, we do not propose changes at this moment.

Section §20.5.6. Ideally most of the function object `operator()`s should be `constexpr`; or at least they should be overloaded to provide that functionality. However, we postpone their modification until the details of “concepts” are known.

Section §20.5.7. Ditto.

Section §20.5.7. Ditto.

3 Clause 21: Strings Library

Modify paragraph §21/1 as follows

This clause describes components for manipulating sequences of any `POD literal` (3.9) type. [...]

Section §21.1.3.1. Modify the definition of `char_traits<char>` as follows:

```

template<>
struct char_traits<char> {
    typedef char char_type;
    typedef int int_type;
    typedef streamoff off_type;
    typedef streampos pos_type;
    typedef mbstate_t state_type;

    static void assign(char_type&, const char_type&);
    static constexpr bool eq(const char_type&, const char_type&);
    static constexpr bool lt(const char_type&, const char_type&);
    static int compare(const char_type*, const char_type*, size_t);
    static size_t length(const char_type*);
    static const char_type* find(const char_type*, size_t,
                                const char_type&);
    static char_type* move(char_type*, const char_type*, size_t);
    static char_type* copy(char_type*, const char_type*, size_t);
    static char_type* assign(char_type*, size_t, char_type);

    static constexpr int_type not_eof(const char_type&);
    static constexpr char_type to_char_type(const int_type&);
    static constexpr int_type to_int_type(const char_type&);
    static constexpr bool eq_int_type(const int_type&, const int_type&);
    static constexpr int_type eof();
};

```

Section §21.1.3.2. Modify the definition of `char_traits<char>` as follows:

```

namespace std {
template<>
struct char_traits<char> {
    typedef wchat_t char_type;
    typedef wint_t int_type;
    typedef streamoff off_type;
    typedef wstreampos pos_type;
    typedef mbstate_t state_type;

    static void assign(char_type&, const char_type&);
    static constexpr bool eq(const char_type&, const char_type&);
};

```

```

static constexpr bool lt(const char_type&, const char_type&);
static int compare(const char_type*, const char_type*, size_t);
static size_t length(const char_type*);
static const char_type* find(const char_type*, size_t,
                             const char_type&);
static char_type* move(char_type*, const char_type*, size_t);
static char_type* copy(char_type*, const char_type*, size_t);
static char_type* assign(char_type*, size_t, char_type);

static constexpr int_type not_eof(const char_type&);
static constexpr char_type to_char_type(const int_type&);
static constexpr int_type to_int_type(const char_type&);
static constexpr bool eq_int_type(const int_type&, const int_type&);
static constexpr int_type eof();
};

```

4 Clause 23: Containers library

Section §23.2.1. Modify the definition of the class template array as follows:

```

template<class T, size_t N>
struct array {
    // ...
    constexpr size_type size() const;
    constexpr size_type max_size() const;
    // ...
};

```

Section §23.3.5. Modify the definition of class template bitset as follows:

```

template<size_t N>
class bitset {
    // ...
    constexpr bitset();
    constexpr bitset(unsigned long);
    // ...
    constexpr size_t size();
    // ...
    constexpr bool operator[](size_t) const;
};

```

5 Clause 26: Numerics library

Section §26.3. Modify paragraph §26.3/2 as follows:

The effect of instantiating the template `complex` for any type other than `float`, `double` or `long double` is unspecified. **The specializations `complex<float>`, `complex<double>`, and `complex<long double>` are literal types (3.9).**

Section §26.3.3. Redefine `complex` specializations as literal types:

```
template<> class complex<float> {
public:
    typedef float value_type;

    constexpr complex(float re = 0.0f, float im = 0.0f);
    explicit constexpr complex(econst complex<double>&);
    explicit constexpr complex(econst complex<long double>&);
    constexpr float real() const;
    constexpr float imag() const;
    complex<float>& operator= (float);
    complex<float>& operator+=(float);
    complex<float>& operator-=(float);
    complex<float>& operator*=(float);
    complex<float>& operator/=(float);
    complex<float>& operator=(const complex<float>&);
    template<class X> complex<float>& operator= (const complex<X>&);
    template<class X> complex<float>& operator+=(const complex<X>&);
    template<class X> complex<float>& operator-=(const complex<X>&);
    template<class X> complex<float>& operator*=(const complex<X>&);
    template<class X> complex<float>& operator/=(const complex<X>&);
};

template<> class complex<double> {
public:
    typedef double value_type;

    constexpr complex(double re = 0.0, double im = 0.0);
    constexpr complex(econst complex<float>&);
    explicit constexpr complex(econst complex<long double>&);
    constexpr double real() const;
    constexpr double imag() const;
```

```

complex<double>& operator= (double);
complex<double>& operator+=(double);
complex<double>& operator-=(double);
complex<double>& operator*=(double);
complex<double>& operator/=(double);
complex<double>& operator=(const complex<double>&);
template<class X> complex<double>& operator= (const complex<X>&);
template<class X> complex<double>& operator+=(const complex<X>&);
template<class X> complex<double>& operator-=(const complex<X>&);
template<class X> complex<double>& operator*=(const complex<X>&);
template<class X> complex<double>& operator/=(const complex<X>&);
};

template<> class complex<long double> {
public:
    typedef long double value_type;

    constexpr complex(long double re = 0.0L, long double im = 0.0L);
    constexpr complex(const complex<float>&);
    constexpr complex(const complex<double>&);
    constexpr long double real() const;
    constexpr long double imag() const;
    complex<long double>& operator= (long double);
    complex<long double>& operator+=(long double);
    complex<long double>& operator-=(long double);
    complex<long double>& operator*=(long double);
    complex<long double>& operator/=(long double);
    complex<long double>& operator=(const complex<long double>&);
    template<class X> complex<long double>& operator= (const complex<X>&);
    template<class X> complex<long double>& operator+=(const complex<X>&);
    template<class X> complex<long double>& operator-=(const complex<X>&);
    template<class X> complex<long double>& operator*=(const complex<X>&);
    template<class X> complex<long double>& operator/=(const complex<X>&);
};

```

Section §26.3.7. At this moment, we do not suggest any change to complex value operations. We postpone such decision to after “concepts” details are known and adoption of `complex<T>` where T is an integral type.

6 Clause 27: Input/Output library

We propose that all bitmask types be defined as enumerations and the appropriate mask operators be overloaded, as constexpr functions. Below, we give an example for `fmtflags`.

Section §27.4.1. Define `ios_base::fmtflags` as an enumeration with constexpr overloaded operators as follows:

```
enum fmtflags {
    boolalpha = implementation-defined value,
    dec = implementation-defined value,
    fixed = implementation-defined value,
    hex = implementation-defined value,
    internal = implementation-defined value,
    left = implementation-defined value,
    oct = implementation-defined value,
    right = implementation-defined value,
    scientific = implementation-defined value,
    showbase = implementation-defined value,
    showpoint = implementation-defined value,
    showpos = implementation-defined value,
    skipws = implementation-defined value,
    unitbuf = implementation-defined value,
    uppercase = implementation-defined value,
    adjustfield = implementation-defined value,
    basefield = implementation-defined value,
    floatfield = implementation-defined value,
};

constexpr fmtflags operator~(fmtflags f)
{
    return fmtflags(~(f));
}

constexpr fmtflags operator&(fmtflags lhs, fmtflags rhs)
{
    return fmtflags(int(lhs) & int(rhs));
}
```

```
constexpr fmtflags operator|(fmtflags lhs, fmtflags rhs)
{
    return fmtflags(int(lhs) | int(rhs));
}

constexpr fmtflags operator^(fmtflags lhs, fmtflags rhs)
{
    return fmtflags(int(lhs) ^ int(rhs));
}
```

Define `ios_base::iostate`, `ios_base::openmode`, and `ios_base::seekdir` similarly.

7 Clause 28: Regular expressions library

Section §28.5. Define regex constant types `regex_constants::syntax_option_type`, `regex_constants::match_flag_type` as literal type, an enumeration with `constexpr` overloaded operators.

Define the regex constants as `constexpr` values.

Define the regex error constant as `constexpr` values.

Section §28.8.1. Define the `basic_regex` constants as `constexpr` values.