Doc no: N1980=06-0050
Date: 2006-04-24
Reply-To: Gabriel Dos Reis
gdr@cs.tamu.edu

Generalized Constant Expressions — Revision 3

Gabriel Dos Reis Bjarne Stroustrup

Texas A&M University

Abstract

This paper generalizes the notion of *constant expressions* to include *constant-expression functions* and *user-defined literals*. In addition, some floating-point constant expressions are allowed. The goal is to improve support for generic programming, systems programming, and library building, and to increase C99 compatibility. The proposal allows us to remove long-standing embarassments from some Standard Library components (notably `<limits>`).

Introduction

This paper generalizes the notion of constant expressions to include calls to "sufficiently simple" functions (*constant-expression functions*) and objects of user-defined types constructed from "sufficiently simple" constructors (*constant-expression constructors*.) The proposal aims to

- improve type-safety and portability for code requiring compile time evaluation;
- improve support for systems programming, library building, generic programming; and
- remove embarassments from existing Standard Library components.

The suggestions in this proposal directly build on previous work — in particular *Generalized Constant Expressions* [DRS06, DR03] and *Literals for user-defined types* [Str03] — and discussions at committee meetings — in particular in Kona (October 2003), Redmond (October 2004), Mont Tremblant (October 2005), and Berlin (April 2006).

1 Problems

Most of the problems addressed by this proposal have been discussed in previous papers, especially the initial proposal for *Generalized Constant Expressions* [DR03], the proposal for *Literals for user-defined types* [Str03], *Generalized initializer lists* [DRS03], *Initializer lists* [SDR05]. What follows is a brief summary.

1.1 Embarassments with numeric limit constants

The standard `numeric_limits` class template provides uniform syntax to access functionality of `<limits.h>`, but fails to deliver constant expressions. For example, the expression `numeric_limits<int>::max()` while functionally equivalent to the macro `INT_MAX`, is not an integral constant. That is due to an unnecessarily restrictive notion of constant expressions. The result is that macros are preferred in situations where values need to be known at compile time.

The main thrust of this proposal suggests to allow explicitly identified simple functions to be used as part of constant expressions.

1.2 Convoluted bitmask types

The Standard Library [ISO03, §17.3.2.1.2] uses the notion of *bitmask type* described as follows:

> 1 Several types defined in clause 27 are *bitmask types*. Each bitmask type can be implemented as an enumerated type that overloads certain operators, as an integer type, or as a bitset (23.3.5).
>
> 2 The bitmask type *bitmask* can be written:

```
enum bitmask {
  V0 = 1 << 0, V1 = 1 << 1, V2 = 1 << 2, V3 = 1 << 3, .....
};
static const bitmask C0(V0);
static const bitmask C1(V1);
static const bitmask C2(V2);
static const bitmask C3(V3);
.....
bitmask operator&(bitmask X, bitmask Y)
        // For exposition only.
        // int_type is an integral type capable of
        // representing all values of bitmask
{ return static_cast<bitmask>(
             static_cast<int_type>(X) &
             static_cast<int_type>(Y)); }
// ...
```

> 3 Here, the names `C0`, `C1`, etc. represent *bitmask elements* for this particular bitmask type. All such elements have distinct values such that, for any pair `Ci` and `Cj`, `Ci&Ci` is nonzero and `Ci&Cj` is zero.

None of the implementation techniques suggested in the C++ standard text is really satisfactory. We are forced to choose between type safety ("elegance") and compile-time evaluation ("efficiency"). For example, if a bitmask type is implemented by an enumeration type with overloads of the appropriate operators, then the masking operators no longer deliver constant expressions when the inputs are constant expressions. That is a real efficiency problem for some system programs. On the other hand, if a bitmask is implemented by an integer type or we rely on the implicit conversion of enumerations to int, then the masking operators come for free and are efficient; but the operators do not provide any type guarantees.

This proposal allow efficient implementation of bitmask type, and without loss of type information.

1.3 Brittle enumerated types

The Standard Library [ISO03, §17.3.2.1.1] uses the notion of *enumerated type* defined as follows:

> 1 Several types defined in clause 27 are *enumerated types*. Each enumerated type may be implemented as an enumeration or as a synonym for an enumeration[150)].

[with footnote 150]

> Such as an integer type, with constant integer values (3.9.1).

> 2 The enumerated type *enumerated* can be written:
>
> ```
> enum enumerated { V0, V1, V2, V3, ... };
>
> static const enumerated C0(V0);
> static const enumerated C1(V1);
> static const enumerated C2(V2);
> static const enumerated C3(V3);
> ....
> ```
>
> 3 Here, the names `C0`, `C1`, etc. represent *enumerated elements* for this particular enumerated type. All such elements have distinct values.

This definition does not prevent user errors, such as accidental use of implicit conversions and operations on the underlying integer type (`operator|`, `operator&`, etc.) Our proposal for literals of user-defined types, combined with constant-expression functions, provide an alternative.

1.4 Unexpected dynamic initialization

In current C++, a variable or static data member declared `const` can be used in an integral constant expression, provided it is of integral type and initialized with constant expression. Similarly, global variables can be statically initialized with

constant expressions. However, it is possible to be surprised by expressions that (to someone) "look `const`" but are not. For example in

```
struct S {
    static const int size;
};

const int limit = 2 * S::size;   // dynamic initialization
const int S::size = 256;
const int z = numeric_limits<int>::max(); // dynamic initialization
```

Here, `S::size` is indeed intialized with a constant expression, but that initialization comes "too late" to make `S::size` a constant expression; consequently `limit` may be dynamically initialized. The issue here is that there is no simple, systematic, and reliable way of requesting that a datum be initialized before its use and the initializer must be a constant exprssion. That problem is addressed using constant-expression values (§2.2).

1.5 Complex rules for simple things

The focus of this proposal is to address the issues mentioned in preceding sections. However, discussions in the Core Working Group at the Berlin meeting (April 2006) concluded that the current rules for integral constant expressions are too complicated, and source of several Defect Reports. Consequently, a "cleanup", *i.e.* adoption of simpler, more general rules is suggested.

2 Suggestions for C++0x

The generalization we propose are articulated in three steps: First, we introduce *constant-expression functions* and use those to generalize constant expressions. Second, we introduce "literals for user-defined type" based on the notion of *constant-expression constructors*. Finally, we describe floating-point constant expressions.

2.1 Constant-expression functions

A function is a *constant-expression function* if

- it returns a value (*i.e.*, has non-`void` return type);
- its body consists of a single statement of the form

  `return` *expr*`;`

  where after substitution of constant expression for the function parameters in *expr*, the resulting expression is a constant expression (possibly involving calls of previously defined constant expression functions); and
- it is declared with the keyword `constexpr`.

This is an elaborate way of saying that a constant-expression function is a named constant expression with parameters, and has been explicitly identified as such. Expressions having the same properties as *expr* above are called *potential constant expressions*. A constant-expression function cannot be called before it is defined.

A constant-expression function may be called with non-constant expressions, in that case there is no requirement that the resulting value be evaluated at compile-time. Here are some examples

```
constexpr int square(int x)
{ return x * x; }              // fine

constexpr long long_max()
{ return 2147483647; }         // fine

constexpr int abs(int x)
{ return x < 0 ? -x : x; }     // fine

constexpr void f(int x)        // error: return type is void
{ /* ... */ }

constexpr int next(int x)
{ return ++x; }                // error: use of increment

constexpr int g(int n)         // error: body not just ``return expr''
{
   int r = n;
   while (--n > 1) r *= n;
   return r;
}

constexpr int twice(int x);
enum { bufsz = twice(256) };   // error: twice() isn't (yet) defined

constexpr int fac(int x)
{ return x > 2 ? x * fac(x - 1) : 1; } // error: fac() not defined
                                       // before use

template<typename T>
  constexpr int bytesize(T t)
  { return sizeof (t); }       // fine


float array[square(9)];        // OK -- not C99 VLA
enum { Max = long_max() };     // OK
bitset<abs(-87)> s;             // OK
extern const int medium;
const int high = square(medium); // OK -- dynamic initialization
char buf[bytesize(0)];         // OK -- not C99 VLA
```

Here "fine" indicates that the function body is simple enough to be evaluated as a constant expression given constant expression arguments.

Note that constant-expression functions provide what we usually expect from functional macros combined with usual pass-by-value evaluation (e.g. the argument to `square` is used twice, but evaluated only once) and type safety. The requirement that a constant-expression function can only call previously defined constant-expression functions ensures that we don't get into any problems related to recursion. Experimental implementations of calls to functions in constant expressions in C++ have long history going back to early versions of CFront.

We (still) prohibit recursion in all its form in constant expressions. That is not strictly necessary because an implementation limit on recursion depth in constant expression evaluation would save us from the possibility of the compiler recursing forever. However, until we see a convincing use case for recursion, we don't propose to allow it.

A constant expression function must be defined before its first use. For example:

```
struct S {
    constexpr int twice();
    constexpr int t();
private:
    static constexpr int val;  // constexpr variable
};

constexpr int S::val = 7;
constexpr int S::twice() { return val + val; }

constexpr S s = { };
int x1 = s.twice();     // ok
int x2 =  s.t();        // error: S::t() not defined

constexpr int ff();     // ok
constexpr int gg();     // ok

int x3 = ff();          // error: ff() not defined

constexpr int ff() { return 1; }        // too late
constexpr int gg() { return 2; }

int x4 = gg();  // ok
```

2.2 Constant-expression data

A *constant-expression value* is a variable or data member declared with the `constexpr` specifier. A *constant-expression value* must be initialized with a constant expression or an rvalue constructed by a constant expression constructor with constant expression arguments. For example:

```
const double mass = 9.8;
constexpr double energy = mass * square(56.6); // OK
extern const int side;
constexpr int area = square(side);  // error: square(side) is not a
                                    //   constant expression
```

A variable or data member declared with `constexpr` behaves as if it was declared with `const`, except that it requires initialization before use and its initializer must be a constant-expression. Therefore a `constexpr` variable can always be used as part of a constant expression.

As for other `const` variables, storage need not be allocated for a constant-expression datum, unless its address is taken. For example:

```
constexpr double x = 9484.748;
const double* p = &x;           // the &x forces x into memory
```

2.3 Constant-expression constructors

The notion of constant-expression data generalizes from data with built-in types to data with user-defined types. To construct constant-expression values of user-defined type, one needs the notion of *constant-expression constructor*: a constructuor

- declared with the `constexpr` specifier;
- with member-initializer part involving only potential constant-expressions; and
- and the body of which is empty.

A constant-expression constructor is just like a constant-expression function, except that since constructors do not return values their body must be empty and the constant expression evaluation happens in member initializations which must deliver constants if the inputs are constants. An object of user-defined type constructed with a constant-expression constructor and constant expression arguments is called a *user-defined literal*. For example:

```
struct complex {
   constexpr complex(double r, double i) : re(r), im(i) { }

   constexpr double real() { return re; }
   constexpr double imag() { return im; }

private:
   double re;
   double im;
};

constexpr complex I(0, 1);  // OK -- literal complex
```

For a constant-expression constructor:

- the definition is checked for consistency with potential constant expression assumptions. It is an error if the definition does not meet those constraints. A constant-expression constructor is `inline`;
- the use with constant expression arguments is guaranteed to yield a user-defined literal, e.g. an expression with user-defined type that is evaluated at compile time.

A constant-expression constructor may be invoked with non-constant expression arguments — the resulting initialization may then be dynamic. This implies that there is no need to have two versions for constructors that would do the same thing, e.g. one constructor that accepts only constant expression arguments and one that may accept non-constant expression arguments. For example:

```
double x = 1.0;
constexpr complex unit(x, 0); // error: x non-constant
const complex one(x, 0);   // OK, ``ordinary const'' -- dynamic
                           //   initialization

constexpr double xx = I.real();  // OK
complex z(2, 4);           // OK -- ordinary variable
```

When the initializer for an ordinary variable (*i.e.* not a `constexpr`) happens to be a constant, the compiler can choose to do dynamic or static initialization (as ever).

Declaring a constructor `constexpr` will help compilers to identify static initialization and perform appropriate optimizations (like putting literals in read-only memory.) Note that since "ROM" isn't a concept of the C++ Standard and what to put into ROM is often a quite subtle design decision, this proposal simply allows the programmer to indicate what might be put into ROM (constant-expression data) rather than trying to specify what actually goes into ROM in a particular implementation.

Using the value of an object declared `constexpr` requires the compiler to "remember" its value for use in constant expressions (later in the same translation unit), like is the case for enumerators. For example:

```
constexpr complex v[] = {
     complex(0, 0), complex(1, 1), complex(2, 2)
};
constexpr double x = v[2].real();  // OK
```

Clearly, a compiler might have to "remember" a lot of values, but then memories on systems running compilers tend to be correspondingly large these days. Also, this kind of "compile-time data bloat" can occur only as the result of explicit use of `constexpr` for large arrays.

Note also that `constexpr` values are those that the compiler can evaluate at translation time. In particular, given

```
constexpr int i = 98;
```

the following declaration is ill-formed

```
const int p = (int) &i;   // ERROR
```

because the initializer is not an integral constant expression.

2.3.1 Destructor

Can an user-defined literal be destroyed? Yes. The destructor needs to be trivial. The reason is that a constant-expression is intended to be evaluated by the compiler at translation time just like any other literal of built-in type; in particular no observable side-effect is permitted. Since destructors do not yield values, the only effect they may have is to modify the state of the (executing) environment. Consequently, to preserve behaviour, we require that the destructor for a user-defined literal be trivial.

2.3.2 Copy-constructor

When a user-defined literal is copied, e.g. arguments passing in function call, using a copy constructor and the copy constructor is trivial, then the copy is also a user-defined literal. For example:

```
constexpr complex operator+(complex z, complex w)
{
  return complex(z.real() + w.real(), z.imag() + w.imag());  // fine
}
constexpr complex I2 = I + I;                 // OK

struct resource {
  int id;
  constexpr resource(int i) : id(i) { }       // fine
  resource(const resource& r) : id(r.id)
  {
    cout << id << " copied" << endl;
  }
};

constexpr resource f(resource d)
{ return d; }                 // error: copy-constructor not trivial

constexpr resource d = f(9);   // error: f(9) not constant expression
```

2.4 Floating-point constant expressions

Traditionally, evaluation of floating-point constant expression at compile-time is a torny issue. For uniformity and generality, we suggest to allow constant-expression

data of floating point types, initialized with any floating-point constant expressions. That will also increase compatibility with C99 [ISO99, §6.6] which allows

> [#5] An expression that evaluates to a constant is required in several contexts. If a floating expression is evaluated in the translation environment, the arithmetic precision and range shall be at least as great as if the expression were being evaluated in the execution environment.

For example, in

```
constexpr complex w = I + complex(3.5, 8.7);           // OK
```

the variable `w` is as if initialized with `complex(3.5, 9.7)`.

2.5 Changes to the C++ standard

The original proposal [DR03] for generalizing constant expressions did not introduce a new keyword to distinguish constant-expression functions from others. That proposal relied on the compiler recognizing such functions being simple enough for use in constant expression. However, during discussions in the Evolution Group at the Kona meeting (October 2003), the consensus was that we needed syntactic marker. Given that (our proposed **`constexpr`**), a programmer can state that a function is intended to be used in a constant expression and the compiler can diagnose mistakes. We considered this in conjunction with the user-defined literal and initializer-list proposals [Str03, SDR05]. At the Mont Tremblant meeting (October 2005), the Evolution Group agreed on the new declaration specifier `constexpr`, for defining constant-expression functions and constants of user-defined types.

The remaining subsections provide necessary wordings to implement the design outlined in the previous sections.

2.5.1 Syntax

New keyword Add the new keyword `constexpr` to "Table 3" [ISO03, §2.11].

New specifier The keyword `constexpr` is a declaration specifier; modify the grammar in [ISO03, §7.1] as follows:

> 1 The specifiers that can be used in a declaration are
>
> *decl-specifier:*
> *storage-class-specifier*
> *type-specifier*
> *function-specifier*
> `friend`
> `typedef`
> **`constexpr`**

We do not propose to make `constexpr` a *storage-class-specifier* because it can be combined with either `static` or `extern` or `register`, much like `const`. We do not propose to make `constexpr` part of *type-specifier* as a *cv-qualifier* because it is not a new distinct type qualifier, and we don't see a need to distinguish between, say, a type for literal `int`, and a separate type for non-literal `int`. That helps keep the type rules as simple as possible. Finally, we do not propose to make `constexpr` a *function-specifier* because it can be used to define both functions and variables. We don't propose to make `constexpr` applicable to function arguments because it would be meaningless for non-inline functions (the argument would be a constant, but the function wouldn't know which) and because it would lead to complications of the overloading rules (can I overload on `constexpr`-ness? — no).

2.5.2 Semantics

New section Add the following section for the description of `constexpr` semantics:

7.1.6 The `constexpr` specifier [decl.constexpr]

1 The `constexpr` specifier can be applied only to names of objects, functions, and function templates. [*Note:* Function parameters cannot be declared `constexpr`.]

2 An entity declared with `constexpr` shall be initialized (if it is an object) or defined (if it is a function) before use.

3 A `constexpr` specifier used in a function or constructor declaration declares that function or constructor to be a *constant-expression function* or a *constant-expression constructor*, respectively. Such a function or constructor is implicitly `inline`.

4 The definition of a constant-expression function shall satisfy the following constraints:

- — its return-type shall not be `void`; and
- — its *function-body* shall be a *compound-statement* of the form

  `{ return` *expression*`; }`

  where *expression* is a potential constant expression (5.19).

A constant-expression function shall not be declared virtual (10.3).

5 The definition of a constant-expression constructor shall satisfy the following constraints:

- — its *function-body* is an empty *compound-statement*; and
- — its *ctor-initializer* initializes data members and base class subobjects using only constant-expression constructors and potential constant expressions.

A trivial constructor is also considered a constant-expression constructor.

6 A `constexpr` specifier used in a nonstatic member function declaration declares that member function to be `const`. The program is ill-formed

if the class-type of which that function is a nonstatic member is not a literal type (3.9).

7 A `constexpr` specifier used in an object declaration declares it as `const`. The object shall be initialized with a potential constant expression.

Paragraph extension Extend paragraph §3.19/10:

10 Arithmetic types (3.9.1), enumeration types, pointers types, and pointer to member types (3.9.2), and *cv-qualified* version of these types (3.9.3) are collectively called *scalar types*. Scalar types, POD-struct types, POD-union types (clause 9), arrays of such types and *cv-qualified* versions of these types (3.9.3) are collectively called POD types. **POD types and non-POD class-types (clause 9) with at least one constant-expression constructor (7.1.6) and trivial destructor (12.4) are collectively called *literal types*.**

New paragraph Add a new paragraph to section §10.1 as follows

7 A class with virtual base class shall not have constant-expression constructors or nonstatic `constexpr` member functions. In particular, a class with virtual base class cannot be a literal type (3.9).

Paragraph modification Modify paragraph §3.6.2/1 as follows:

1 Objects with static storage duration (3.7.1) shall be zero-initialized (8.5) before any other initialization takes place. Zero-initialization and initialization with a constant expression are collectively called *static initialization*; all other initialization is *dynamic initialization*. Objects of ~~POD~~ **literal** types (3.9) with static storage duration initialized with constant expressions (5.19) shall be initialized before any dynamic initialization takes place. Objects with static storage duration defined in namespace scope in the same translation unit and dynamically initialized shall be initialized in the order in which their definition appears in the translation unit. [*Note:* 8.5.1 describes the order in which aggregate members are initialized. The initialization of local static objects is described in 6.7.]

Paragraph modification Modify paragraph §9.2/4 as follows:

4 A *member-declarator* can contain a *constant-initializer* only if it declares a `static` member (9.4) of `const` ~~integral or `const` enumeration~~ **literal** types, see 9.4.2.

2.5.3 Constant expressions revised

Paragraph modification Replace paragraph §5.19/1 with:

1 Certain contexts require expressions that satisfy additional requirements as detailed in this subclause. Such expressions are called constant expressions. [*Note:* Those expressions can be evaluated during translation.] An expression is a *constant expression* if and only if it is

— a literal (2.13), or

— an *id-expression* that refers to an enumerator, a non-type template parameter or a non-volatile const variable or non-volatile const static data member of literal type (3.9) initialized with a constant expression (8.5), or

— it is a `sizeof` expression (5.3.3), or

— an invocation of a built-in constant-expression operator (clause 5) or a constant-expression constructor or constant-expression function (7.1.6), where all arguments are constant expressions [*Note:* Overload resolution (13.3) is applied as usual.], or

— an invocation of a trivial copy-constructor of literal type, with a constant expression argument, or

— `*this` within a `constexpr` member function definition [*Note:* `this` is not a constant expression.].

Paragraph modification Modify paragraph §5.19/2 as follows:

2 Other expressions are considered *constant-expression*s only for the purpose of non-local static object initialization (3.6.2). Such constant expressions shall evaluate to one of the following:

— a null pointer value (4.10),

— a null member pointer value (4.11),

— ~~an arithmetic constant expression,~~

— an address constant expression,

— a reference constant expression,

— an address constant expression for a complete object type, plus or minus an integral constant expression, or

— a pointer to member constant expression.

New paragraph Add a new paragraph to the section §5.19:

7 An expression appearing in a *function-body* of a constant-expression function or constant-expression constructor is a *potential constant expression* if it is a constant expression when all occurences of function parameters are replaced by arbitrary constant expressions of the appropriate type.

Paragraph removal Remove paragraph §5.19/3 because it is no longer needed.

2.5.4 Address constant expressions

Paragraph modification Modify paragraph [ISO03, §5.19]:

8 An *address constant expression* is a pointer to an lvalue designating an object of static storage duration, a string literal (2.13.4), or a function. The pointer shall be created explicitly, using the unary `&` operator, or implicitly using a non-type template parameter of pointer

type, **or invoking constant-expression constructors or constant-expression functions with constant expression arguments,** or using an expression of array (4.2) or function (4.3) type. The subscripting operator `[]` and the class member access `.` and `->` operators, the `&` and `*` unary operators, and pointer casts (except `dynamic_casts`, 5.2.7) can be used in the creation of an address constant expression, but the value of an object shall not be accessed by the use of these operators. If the subscripting operator is used, one of its operands shall be an integral constant expression. An expression that designates the address of a subobject of a ~~non-POD~~ **non-literal** class object (clause 9) is not an address constant expression (12.7). Function calls shall not be used in an address constant expression, even if the function is `inline` and has a reference return type **unless that function is a constant-expression function invoked with constant expression arguments**.

Note that while the result of calling a `constexpr` member function with constant expression arguments (including the implied object) is a constant expression, the value of `this` is not considered an address constant expression within the body of the `constexpr` function.

2.5.5 Reference constant expressions

Paragraph modification Modify paragraph [ISO03, §5.19]:

9 A *reference constant expression* is an lvalue designating an object of static storage duration, a non-type template parameter of reference type, **or invoking explicitly or implicitly constant-expression constructors or functions with constant expression arguments,** or a function. The subscripting operator `[]`, the class member access `.` and `->` operators, the `&` and `*` unary operators, and reference casts (except those invoking **non-constant expression** user-defined conversion functions (12.3.2) and except `dynamic_casts` (5.2.7)) can be used in the creation of a reference constant expression, but the value of an object shall not be accessed by the use of these operators. If the subscripting operator is used, one of its operands shall be an integral constant expression. An lvalue expression that designates a member or base class of a ~~non-POD~~ **non-literal** class object (clause 9) is not a reference constant expression (12.7). Function calls shall not be used in a reference constant expression, even if the function is `inline` and has a reference return type **unless that function is a constant-expression function invoked with constant expression arguments**.

2.5.6 Other changes

New paragraphs Add new paragraphs to clause 5:

11 A *constant-expression operator* is one of

- — the built-in unary operators `*`, `+`, `-`, `&`, `!`, `~`,
- — the cast operators `static_cast`, `const_cast`, `reinterpret_cast`,
- — the subscription operator `[]`,
- — the member access operator,

— the built-in binary operators `*`, `/`, `%`, `+`, `-`, `<<`, `>>`, `<`, `>`, `<=`, `>=`, `==`, `!=`, `&`, `^`, `|`, `&&`, `||`,

— the ternary conditional operator `?:`.

[*Note:* The comma operator is not a constant-expression operator.]

The member access operator yields a constant expression if and only if the access is to a non-`mutable` data member with non-`volatile` literal type, and the containing object is the result of a constant expression.

12 Constant-expression operators yield constant expressions only where their operands are themselves constant expressions or literal types. More stringent requirements are placed on each individual operation as described below.

Paragraph modification Modify paragraph §3.2/2 as follows:

2 An expression is *potentially evaluated* unless it appears where ~~an integral~~ **a** constant expression is required (see 5.19), is the operand of the `sizeof` operator (5.3.3), ...

The rationale for this modification is to allow constant expressions of literal types too.

Paragraph modification Modify paragraph §3.2/2 as follows:

5 ... except that a name to a `const` object with internal or no linkage if the object has the same ~~integral or enumeration~~ **literal** type in all definitions of `D`, and the object is initialized with a constant expression (5.19), ...

Paragraph modification Modify paragraph §6.7/4 as follows:

4 .... A local object of ~~POD~~ **literal** type (3.9) with static storage duration initialized with a *constant-expression* is initialized before its block is first entered. ....

Paragraph modification Modify paragraph §9/4 as follows:

4 If a `static` data member is of `const` ~~integral or `const` enumeration~~ **literal** type, its declaration in the class definition can specify a *constant-expression* which shall be ~~an integral~~ **a literal** constant expression (5.19). In that case the member can appear in ~~integral~~ constant expressions. The member shall still be defined in a namespace scope if it is used in the program and the namespace scope definition shall not contain an *initializer*.

Paragraph modification Modify paragraph §14.6.2.3/1 as follows:

2 An *identifier* is value-dependent if it is:

— a name declared with a dependent type,

— the name of a non-type template parameter,

— a constant with ~~integral or enumeration~~ **literal** type and is initialized with an expression that is value-dependent.

....

3 Related proposals

3.1 Standard Library changes

We plan to propose changes to the standard library to take advantage of `constexpr`. Obvious candidates are `numeric_limits`, `bitmask`, and `enumerated` as descibed in §1 and `initializer_list`.

3.2 Non-type template parameter

The suggestion of extending non-type template parameter type to literal types will be subject of an independent proposal.

3.3 Generalizing PODs

There is a suggestion to extend the notion of POD. That suggestion is independent, in scope, of this constant expression proposal. The definition of "literal type" as suggested in this paper may be a starting point for that proposal.

4 Acknowledgments

Thanks to the committee members who provided feedback, suggestions for improvement, as expressed in face-to-face meetings or on the standard reflectors.

References

[DR03] Gabriel Dos Reis. Generalized Constant Expressions. Technical Report N1521=03-0104, ISO/IEC JTC1/SC22/WG21, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2003/n1521.pdf`, September 2003.

[DRS03] Gabriel Dos Reis and Bjarne Stroustrup. Generalized initializer list. Technical Report N1509=03-0092, ISO/IEC JTC1/SC22/WG21, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2003/n1509.pdf`, September 2003.

[DRS06] Gabriel Dos Reis and Bjarne Stroustrup. Generalized Constant Expressions – Revision 2. Technical Report N1972=06-0042, ISO/IEC SC22/JTC1/WG21, February 2006. Supersedes [DR03].

[ISO99] International Organization for Standards. *International Standard ISO/IEC 9899. Programming Languages — C*, 1999.

[ISO03] International Organization for Standards. *International Standard ISO/IEC 14882. Programming Languages — C++*, 2nd edition, 2003.

[SDR05] Bjarne Stroustrup and Gabriel Dos Reis. Initializer lists. Technical Report N1919=05-0179, ISO/IEC JTC1/SC22/WG21, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2005/n1919.pdf`, December 2005.

[Str03] Bjarne Stroustrup. Literals for user-defined types. Technical Report N1511=03-0094, ISO/IEC JTC1/SC22/WG21, `http://www.open-std.org/JTC1/SC22/WG21/docs/papers/2003/n1511.pdf`, September 2003.