Doc. no. N1521=03-0104
Date: September 21, 2003
Reply-To: Gabriel Dos Reis
gdr@acm.org

Generalized Constant Expressions

Abstract

We suggest to generalize the notion of *constant expressions* to include calls to *constant-valued* functions. The purpose is to push their expressive power even further, to remove embarassments in the standard library, to provide some support for meta programming and convenient notation for expressions that are morally constant.

Introduction

This paper proposes to generalize the definition of constant expressions to include calls to suitable simple functions with constant expressions — which abstractly are constant expressions with named sub-patterns. It aims at providing better type-safety support for components of the standard library (or general libraries), to remove embarrassments, and to enhance the expressive power of constant expressions. The suggestions contained in this paper are not intended to be final wordings. Rather, they are initial basis for discussions and improvements.

1 The problems

This section describes examplars of problems the idea of generalizaing constant expressions, as proposed in §2.2, is trying to solve. At the end of this section, we recall the definition of constant expressions as currently in use.

1.1 Non-portable, non-compile time bitmasks

The Standard Library introduces the notion of *bitmask types* in its introductory clause (see [1, §17.3.2.1.2]):

> 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).

For type-safety reasons, it is highly desirable that such a bitmask type be implemented by a type different from any standard integer type. Enumeration types are popular choice. However, once the aforementioned "certain operators" have been overloaded, bitmask operations involving only constants no longer yields constant expressions.

```
enum fmtflags {
  boolapha, dec, fixed, hex, internal, left, oct, right,
   // ...
};

inline fmtflags
operator|(fmtflags a, fmtflags b)
{ return fmtflags(int_type(a) | int_type(b)); }
// ...

const fmtflags
  adjustfield = left | right | internal // NOT a constant
```

Because the operator `|` is overloaded, `adjustfield` is not longer a constant. The depenency in implementation defined semantics is another level of non-portability but the core of the problem as just explained is general. Furthermore, there results a gratuitous discrepency between the bitmask `adjustfield` and other enumerations (e.g. `boolapha`). On the other hand it is clear that the function `operator|` is sufficiently simple so that `adjustfield` can be considered a constant, even though only a restrictive definition of constant expression makes it not a constant.

1.2 Embarassments with numeric constants

The Standard Library defines a traits (`numeric_limits`) that provides C++ programs with information about various properties of the implementation's representation of fundamental arithmetic types. For example, `numeric_limits<T>::is_signed` is an integral constant expression that evaluates to `true` when the type `T` is signed. If `T` is an integer type, then associated macros `XXX_MIN` and `XXX_MAX` (defined in `<climit>`) are integral constant expressions that denotes the minimum and maximum values of `T`. The same values are available as calls to functions

`numeric_limits<T>::min()` and `numeric_limits<T>::max()`, except that they are no longer integral constant expressions, mostly because of a restrictive definition of constant expression.

1.3 A zoo of constant expressions

Standard C++ defines a broad notion of constant expressions (see [1, §5.19]). A *constant-expression* is a *conditional-expression* subject to specific restrictions. The Standard defines eight categories of constant-expressions

1. an *integral constant expression*,
2. a *null pointer value*,
3. a *null member pointer value*,
4. an *arithmetic constant expression*,
5. an *address constant expression*,
6. a *reference constant expression*,
7. an address constant expression for a complete object type, plus or minus an integral constant expression, or
8. a *pointer to member constant expression*.

Integral constant expressions are probably the most used constant expressions in various compile-time computations including array definitions, or explicit values for non-type template parameters. For future references, the remaining of this sub-section recalls definitions for the categories of constant expressions listed above. References contained in those definitions are with respect to the Standard [1].

Integral constant expression

> An *integral constant-expression* can involve only literals (2.13), enumerators, *const* variables or static data members of integral or enumeration types initialized with constant expressions (8.5), non-type template parameters of integral or enumeration types, and *sizeof* expressions. Floating literals (2.13.3) can appear only if they are cast to integral or enumeration types. Only type conversions to integral or enumeration types can be used.

> In particular, except in *sizeof* expressions, functions, class objects, pointers, or references shall not be used, and assignment, increment, decrement, function-call, or comma operators shall not be used.

The last restriction rules out expressions like `square(9)` with

```
inline int square(int x) { return x * x; }
```

whereas are it accepts constructs like `SQUARE(9)` with

```
#define SQUARE(X) ((X) * (X))
```

That is an unsual situation, and we believe it is untenable given the safety provided by `square`.

Null pointer value

> A *null pointer constant* is an integral constant expression (5.19) rvalue of integer type that evaluates to zero. A null pointer constant can be converted to a pointer type; the result is the *null pointer value* of that type and is distinguishable from every other value of pointer to object or pointer to function type.

Null member pointer value

> A null pointer constant (4.10) can be converted to a pointer to member type; the result is the *null member pointer value* of that type and is distinguishable from any pointer to member not created from a null pointer constant.

It may be noted that neither a null pointer value, nor a null member pointer value are acceptable template arguments.

Arithmetic constant expression

> An *arithmetic constant expression* shall satisfy the requirements for an integral constant expression, except that
>
> - floating literals need not be cast to integral or enumeration type, and
> - conversions to floating point types are permitted.

Address constant expression

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 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_cast`s, 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 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.

Reference constant expression

A *reference constant expression* is an lvalue designating an object of static storage duration, a non-type template parameter of reference type, or a function. The subscripting operator `[]`, the class member access `.` and `->` operators, the `&` and `*` unary operators, and reference casts (except those invoking user-defined conversion functions (12.3.2) and except `dynamic_cast`s (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 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.

Pointer to member constant expression

A *pointer to member constant expression* shall be created using the unary `&` operator applied to a *qualified-id* operand (5.3.1), optionally preceded by a pointer to member cast (5.2.9).

2 Suggestions

The generalizations we propose in this paper are articulated in three steps: First, the key notion of constant-valued function is introduced; then, we move on refining the current definition of constant expressions; finally, we suggest an even more general notion of constant expressions, building on the proposal [2] for user-defined literals. Texts that introduce modification to existing standard wordings are typeset in **bold** face.

2.1 Constant-valued functions

A function is *constant-valued* if

- it is a non-`void` returning inline function; and
- its body consists in a single statement of the form

  `return` *expr* `;`

  where after substitution of constant expressions for the function parameters in *expr*, the resulting expression is a constant expression. Such an expression may involve calls to previously defined constant-valued functions with argument list consisting only in constant expressions.

The above definition is an elaborated way of saying that a constant-valued function is just a convenient notation for a constant expression where some constant sub-expressions are named.

Examples.

```
int square(int x)
{ return x * x; }                // constant-valued

long long_max()
{ return 2147483647; }           // constant-valued

int abs(int x)
{ return x < 0 ? -x : x; }       // constant-valued

int next(int x)
```

```
{ return ++x; }                    // NOT constant-valued

float array[square(9)];                     // OK
enum { Max = long_max() };                  // OK
bitset<abs(-87)>  s;                        // OK
char buf[next(255)];                        // ERROR
enum { default_fmt = adjustfiled };         // OK
```

It may be noted that constant-valued functions implement what one gains with functional macros combined with constant expressions. A constant-valued function cannot be recursive and cannot display mutual recursion. There is no inherent unsolvable difficulty in constant-valued functions implementations. Experimental implementations of this notion of constant-valued functions calls where integral constants are expected were conducted in earlier versions of CFront [3]. Constant-valued functions do not suffer from the problems and shortcomings of macros.

2.2 Generalizing constant expressions

The generalization of constant expressions we propose in this paper builds on generalizing integral constant expressions, address constant expressions and reference constant expressions.

2.2.1 Basic cases

A first straightforward step is to allow calls to constant-valued functions with constant expressions arguments in integral constant expressions.

Generalized integral constant expression

> An *integral constant-expression* can involve only literals (2.13), enumerators, *const* variables or static data members of integral or enumeration types initialized with constant expressions (8.5), non-type template parameters of integral or enumeration types, **constant-valued function-calls** and *sizeof* expressions. Floating literals (2.13.3) can appear only if they are cast to integral or enumeration types. Only type conversions to integral or enumeration types can be used. In particular, except in *sizeof* expressions, functions, class objects, pointers, or references shall not be used, and assignment, increment, decrement, **non-constant-valued function-call**, or comma operators shall not be used.

Generalized address constant

> 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 calling constant-valued functions with constant expressions**, 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_cast`s, 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 class object (clause 9) is not an address constant expression (12.7). **Non-constant-valued function calls shall not be used in an address constant expression**.

Generalized reference constant expression

> A *reference constant expression* is an lvalue designating an object of static storage duration, a non-type template parameter of reference type, or a function. The subscripting operator `[]`, the class member access `.` and `->` operators, the `&` and `*` unary operators, and reference casts (**except those invoking non-constant-valued conversion functions** and except `dynamic_cast`s (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 class object (clause 9) is not a reference constant expression (12.7). **Non-constant-valued function calls shall not be used in a reference constant expression**.

2.2.2 Advanced cases

The next step of generalization builds on the user-defined literals proposal [2]. This slight generalization formalizes the observation that a mem-

ber of a user-defined literal may also be considered a constant expression; in particular, a data member of a user-defined literal may be a constant expression, and invokation of constant-valued member function for a user-defined literal is a constant expression. Data member accesses and constant-valued member function calls for user-defined literals in constant expressions appear to us as a logical necessity to exploit the expressive power of user-defined literals.

Examples

```
struct nil {
   nil() { }               // literal constructor
   template<class T>
     operator T* () const
     { return 0; }         // constant-valued function
};

 const int* p = nil();      // int* null pointer value
```

More general integral constant expression

> An *integral constant-expression* can involve only literals (2.13), enumerators, *const* variables or static data members of integral or enumeration types initialized with constant expressions (8.5), non-type template parameters of integral or enumeration types, **constant-valued function-calls**, **nonstatic data members of integral or enumeration types of user-defined literals** and *sizeof* expressions. Floating literals (2.13.3) can appear only if they are cast to integral or enumeration types. Only type conversions to integral or enumeration types can be used. In particular, except in *sizeof* expressions, functions, **non-literal objects**, pointers, or references shall not be used, and assignment, increment, decrement, **non-constant-valued function-call**, or comma operators shall not be used.

Examples

```
struct cayley {
  const int norm;
  cayley(int a, int b)
      : value(square(a) + square(b)) { }
```

```
    operator int() const { return norm; }
};

bitset<cayley(98, -23)> s; // eq. to bistet<10133> s;
```

It may be observed that, modulo jumping through syntactic whoops, the newly proposed definition for integral constant expression essential provides the same expressive power of the defintion proposed §2.2.1.

More general address constant expression

> 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 calling constant-valued functions with constant expressions**, 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_cast`s, 5.2.7) can be used in the creation of an address constant expression, but the value of a **non-literal 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 class object (clause 9) is not an address constant expression (12.7). **Non-constant-valued function calls shall not be used in an address constant expression**.

Even more general reference constant expression

> A *reference constant expression* is an lvalue designating an object of static storage duration, a non-type template parameter of reference type, or a function. The subscripting operator `[]`, the class member access `.` and `->` operators, the `&` and `*` unary operators, and reference casts (**except those invoking non-constant-valued conversion functions** and except `dynamic_cast`s (5.2.7)) can be used in the creation of a reference constant expression, but the value of a **non-literal 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 class object (clause 9) is not a reference constant expression (12.7). **Non-constant-valued function calls shall not be used in a reference constant expression**.

3 Interactions with other functionalities

3.1 User-defined literals

This proposal is written as a complement to B. Stroustrup's proposal on *literals for user-defined types*. If user-defined literals are combined with member accesses then one recoves some of the functionality proposed in §2.2.1.

3.2 Nullptr

If combined with user-defined literals, then the semantics (not the syntax) of the nullptr proposal may be conveniently approximated. Whether the nullptr proposal should be provided as a purely library functionality is out of the scope of this proposal. However, it is interesting example of the kind of expression power one gains.

3.3 Inlining

This proposal does not require arbitrary inlining per se. It does not make inlining mandatory. What it does is to change the handling of simple inline functions that are used in contexts where constant expressions are expected. However, implementations are still free to ignore inlining requests in non-constant expression context. The suggestions have been worded so as not to require compilers "to solve the halting problem."

4 Implementation

We do not expect any particular problem from implementation point of view. Some implementations like EDG front-ends already have some of the functionalities proposed in this paper. Earlier versions of CFront had

some experimental implementations of those ideas. In a nutshell, this proposal is essentially about liberal use of "constant folding", which is already common practice in most implementations.

Acknowledgements

This proposal has its roots in dicsussions with B. Stroutrup concerning the *Generalized Initialized List* proposal and his proposal on *Literal Constructor*. The generalization of constant expressions proposed in this paper are intended to complement Stroustrup's user-defined literals.

References

[1] International Standard ISO/IEC 14882, *Programming Languages – C++*.

[2] B. Stroustrup, *Literals for user-defined types*.

[3] B. Stroustrup, private communication.