Doc. no: N3039=0029

Date: 2010-02-14

Reply-To: bs@cs.tamu.edu

Constexpr functions with const reference parameters (a summary)

Bjarne Stroustrup **Alisdair Meredith** **Gabriel Dos Reis**

Here is a message sent to CWG last year during the Santa Cruz meeting to document the EWG decision:

```
From Bjarne Stroustrup <bs@cs.tamu.edu> Fri, 23 Oct 2009 11:10:17 EDT
From: Bjarne Stroustrup <bs@cs.tamu.edu>
Date: Fri, 23 Oct 2009 08:10:06 -0700
Subject: EWG approved extension: Allow constexpr for const T& arguments
and

Allow constexpr for const T& arguments and return values.

EWG voted to move it to CWG

Motivation: many reasonable functions (incl. std library functions)
cannot be constexpr for no other reason than the prohibition against
const T& arguments and return value. The most obvious example is

    template<typename T>
         constexpr const T& max(const T&, const T&);

Proposer: Gaby, supported by Beman and Alisdair.


An implementation has existed for more than a year. A version without
"const T&" (the previously voted parts) was sent to Jason Merrill for
review a couple of weeks ago.  The implementation with "const T&" is
ready to ship (as soon as the previous patch is approved).

The recommendation now is to allow const reference parameters, and
const reference return type, as long as the types referred to are
literal.

   * replace paragraph 7.1.5/3 with

      The definition of a constexpr function shall satisfy the
following constraints:

        -- it shall not be virtual
        -- its return type shall be a literal type or a reference to a
           const-qualified literal type
        -- each of its parameter type shall be a literal type or a
           reference to a const-qualified literal type
```

```
   * replace first bullet of 7.1.5/4 with
       -- each of its parameter types shall be a literal type or
          a reference to a const-qualified literal type

Note that general uses of references and pointers are poison for
constant expression evaluation (would more or less turn the compiler
into an interpreter), so we don't propose to lift any of the existing
restrictions for those.
```

Since then, the wording has been improved a bit (see below) and the implementation distributed and examined. The purpose of this note is simply to present the rationale in greater detail as requested by Doug Gregor on the core reflector (2/10/2010) for the benefit of those who (until now) didn't notice the EWG vote or the CWG issue #991 (which refers to the message above). This proposal also addresses National Body comment FR 23.

Why allow const T& arguments?

Generalized constant expressions (constexpr) as proposed and voted into the WP did not allow for references as arguments and return values. This is natural and cautious given the traditional problems that optimizers have with pointers and references. However, the complete design that we had and its implementation did support const reference arguments. After all, from a fundamental point of view, a **const T&** argument is simply a different implementation of a **T** argument (and usually an optimization). Thus, if we can handle a **T** argument at compile time, we can handle a **const T&** argument: we just make a "temporary" value representing it. That was investigated in Gabriel Dos Reis and Bjarne Stroustrup: *General Constant Expressions for System Programming Languages* (SAC-2010. The 25th ACM Symposium on Applied Computing. March 2010) and the GCC implementation bears out that intuition. The mechanism is exactly the one used to handle ***this** to allow **constexpr** member functions.

So we can handle **const T&**, but should we? Are there use cases to make it worthwhile? Yes, often people use **const T&** in preference to **T** in generic code. Allowing the latter but not the former would warp people's code and force an unnecessary difference in programming style between simple and performance critical code (using **constexpr**) and more general code. The key test is the standard library. Which functions would we like to (sometimes) have evaluated at compile time? Obviously, some people would like just about everything evaluated at compile time, but that would require heroic efforts from compiler- and library writers, so nobody is proposing that (except academic paper reviewers☺). Instead, the functions to look at are those that fit the **constexpr** function pattern: naturally implemented by a single return statement. In particular, this eliminates almost all standard library algorithms from consideration because they are loops.

We have

```
constexpr pair();
```

But only (not constexpr):

```
pair(const T1& x, const T2& y);
```

That's illogical and limiting. If both arguments to **pair()** are **constexpr**, so should the **pair**.

The "classical example" is max(). Here are the **min** and **max** functions that don't use variadic templates. All of these are motivating cases:

```
template<class T> const T& min(const T& a, const T& b);
template<class T> const T& min(const T& a, const T& b, const T& c);
template<class T, class Compare>
const T& min(const T& a, const T& b, Compare comp);
template<class T> const T& max(const T& a, const T& b);
template<class T> const T& max(const T& a, const T& b, const T& c);
template<class T, class Compare>
const T& max(const T& a, const T& b, Compare comp);
template<class T> pair<const T&, const T&> minmax(const T& a, const T& b);
template<class T> pair<const T&, const T&> minmax(const T& a, const T& b, const T& c);
template<class T, class Compare>
pair<const T&, const T&> minmax(const T& a, const T& b, Compare comp);
```

Note that the **minmax** versions can be constexpr only provided the **pair** constructor is, and a reference type is considered a literal type. This is an indication of the importance of having the appropriate standard library functions constexpr. On the one hand, the **min** and **max** functions are easy to handle as constexpr. On the other hand, **minmax** functions rely on the constructor **pair()** , and the class **pair<const T&, const T&>**. So, they would have to be constexpr and literal types, respectively. That is a consideration that CWG would have to weight while reviewing the proposed wording.

Relational operators are also candidates for constexpr status.

```
// 20.3.1, operators:
namespace rel_ops {
        template<class T> bool operator!=(const T&, const T&);
        template<class T> bool operator> (const T&, const T&);
        template<class T> bool operator<=(const T&, const T&);
        template<class T> bool operator>=(const T&, const T&);
}
```

Suggested Core Changes

These proposed changes are with respect to N3000. We make sure that the compiler is not required to evaluate something that is not known until link time or run time.

- replace the second and third bullets of paragraph 7.1.5/3 with

  -- its return type shall be a literal type or a reference to a const-qualified literal type

  -- each of its parameter type shall be a literal type or a reference to a const-qualified literal type

- replace first bullet of 7.1.5/4 with

--each of its parameter types shall be a literal type or a reference to a const-qualified literal type

- Add the following sub-bullet to bullet 6 of 5.19/2
  --an lvalue of literal type that refers to a non-volatile temporary object initialized with a constant expression