Document number: N3615
Date: 2013-03-18
Working group: EWG
Reply to: gdr@cs.tamu.edu

Constexpr Variable Templates

Gabriel Dos Reis
Texas A&M University

Abstract

The aim of this proposal is to simplify definitions and uses of parameterized constants. It allows the declaration of constexpr variable templates. The upshot is a simpler programming rule to remember. It supersedes currently known workarounds with more predictable practice and semantics.

1 The Problem

C++ has no notation for parameterized constants as direct as for functions or classes. There are well known workarounds for this problem:

- constexpr static data members of class templates
- constexpr function templates returning the desired values

These workarounds have been known for decades and well documented. Standard classes such as `std::numeric_limits` are archetypical examples. Although these workarounds aren't perfect, their drawbacks were tolerable to some degree because in the C++03 era only simple, builtin types constants enjoyed unfettered direct and efficient compile time support. All of that changed with the adoption of `constexpr` variables in C++11, which extended the direct and efficient support to constants of user-defined types. Now, programmers are making constants (of class types) more and more apparent in programs. So grow the confusion and frustrations associated with the workarounds.

1.1 Constexpr static data members of class templates

The standard class `numeric_limits` is the archetypical example:

```
template<typename T>
   struct numeric_limist {
      static constexpr bool is_modulo = ...;
   };
// ...
template<typename T>
   constexpr bool numeric_limits<T>::is_modulo;
```

The main problems with "static data member" are:

- they require "duplicate" declarations: once inside the class template, once outside the class template to provide the "real" definition in case the constants is odr-used.
- programmers are both miffed and confused by the necessity of providing twice the same declaration. By contrast, "ordinary" constant declarations do not need duplicate declarations.

1.2 Constexpr function templates

Well known examples in this category are probably static member functions of `numeric_limits`, or functions such as `boost::constants::pi<T>()`, etc.

Constexpr functions templates do not suffer the "duplicate declarations" issue that static data members have; furthermore, they provide functional abstraction. However, they force the programmer to chose in advance, at the definition site, how the constants are to be delivered: either by a const reference, or by plain non-reference type. If delivered by const reference then the constants must be systematically be allocated in static storage; if by non-reference type, then the constants need copying. Copying isn't an issue for builtin types, but it is a showstopper for user-defined types with value semantics that aren't just wrappers around tiny builtin types (e.g. matrix, or biint, or bitfloat, etc.) By contrast, "ordinary" const(expr) variables do not suffer from this problem. A simple definition is provided, and the decision of whether the constants actually needs to be layout out in storage only depends on the *usage*, not the definition.

2 Proposed Solution

This proposal makes a very simple suggestion: *allow the definition and uses of constexpr variable templates*. The technical part of the proposal actually consists of relaxing constraints on template declarations.

2.1 Modification to the standard text

Most of the modifications consist of adding "constexpr variable templates" to the list of entities designated by a template-id, etc.

1. Modify paragraph 14/1 to say

   > The declaration in a *template-declaration* shall
   >
   > — declare or define a function or a class or a **constexpr variable**, or
   >
   > [...] A *template-declaration* is a *declaration*. A *template-declaration* is also a definition if its *declaration* defines a function, a class, **a variable**, or a static data member.

2. Modify paragraph 14.3.3/1:

   > A *template-argument* for a template *template-parameter* shall be the name of a class template or an alias template **or constexpr variable template**, expression as *id-expression*. When the *template-argument* names a class template **or a constexpr variable template**, only primary class templates **or constexpr variable template** are considered when matching the template template argument with the corresponding parameter; partial specializations are not considered even if their parameter lists match that of the template template parameter.

3. Modify paragraph 14.3.3/2

   > Any partial specializations (14.5.5) associated with the primary class template **or constexpr variable template** are considered when a specialization based on the template *template-parameter* is instantiated....

4. Modify paragraph 14.3.3/3

A *template-argument* matches a template *template-parameter* (call it P) when each of the template parameters in the *template-parameter-list* of the *template-argument*'s corresponding class template or alias template **or constexpr variable template** (call it A) ....

5. Modify paragraph 14.4/1

   Two *template-id*s refer to the same class or function **or variable** if ...

3 Extensions

This proposal focuses on the more pressing and narrower need for a simpler notation for parameterized constants. It did not escape the attention of the author that a more general notion of variable templates (not necessarily constexpr) could be defined. That greater generality would need more experiments and is therefore not part of this proposal.

4 Conclusion

This proposal aims for a simple extension to C++: allow constexpr variable templates. It makes definitions and uses of parameterized constants much simpler, leading to simplified and more uniform programming rules to teach and to remember.

Acknowledgement

Vicente J. Botet Escriba independently suggested support for constexpr variable templates on the `std-proposals` mailing list.