Document number: N3651
Date: 2013-04-19
Working group: CWG
Reply to: gdr@cs.tamu.edu

Variable Templates (Revision 1)

Gabriel Dos Reis
Texas A&M University
http://www.axiomatics.org/~gdr/

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. For instance, we would like to represent the mathematical constant π with precision dictated by a floating point datatype

```
template<typename T>
  constexpr T pi = T(3.1415926535897932385);
```

and use it in generic functions, e.g. to compute the area of a circle with a given radius:

```
template<typename T>
 T area_of_circle_with_radius(T r) {
    return pi<T> * r * r;
 }
```

The types of variable templates are not restricted to just builtin types; they can be user defined types. For example, here are definitions of fundamental Pauli matrices (used in quantum mechanics):

```
template<typename T>
  constexpr pauli<T> sigma1 = { { 0, 1 }, { 1, 0 } };
template<typename T>
  constexpr pauli<T> sigma2 = { { 0, -1i }, { 1i, 0 } };
template<typename T>
  constexpr pauli<T> sigma3 = { { 1, 0 }, { -1, 0 } };
```

where `pauli<T>` is a 2x2 matrix with entries of type `complex<T>`.

Alas, existing C++ rules do not allow a template declaration to declare a variable. There are well known workarounds for this problem:

- use constexpr static data members of class templates
- use 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.

2 Workarounds

2.1 Constexpr static data members of class templates

The standard class `numeric_limits` is the archetypical example:

```
template<typename T>
   struct numeric_limits {
      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.

2.2 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 integer, or bigfloat, 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.

3 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.

3.1 Syntax

This proposal does not contain any syntax modification. The reason is that the current grammar allows any declaration to be parameterized. The prohibition of variable template declaration is done in prose via semantics constraints.

3.2 Modification to the standard text

Most of the modifications consist of adding "variable templates" to the list of entities designated by a template-id, etc. Otherwise the changes are straightforward — since the extension itself is conceptually simple. Given the repetitive nature of the changes, one wonders if the standard text might benefit from terminological simplification.

1. Modify paragraph 14/1 to say

> The declaration in a *template-declaration* shall
>
> — declare or define a function ~~or~~, a class, or a 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. A declaration introduced by a template declaration of a variable is a *variable template*. A variable template at class scope is a *static data member template*.

Add examples:

> [Example:
>
> ```
> template<typename T>
>   constexpr T pi = T(3.1415926535897932385);
>
> template<typename T>
> T circular_area(T r) {
>    return pi<T> * r * r;
> }
>
> struct matrix_constants {
>   template<typename T>
>     using pauli = hermitian_matrix<T, 2>;
>
>   template<typename T>
>     constexpr pauli<T> sigma1 = { { 0, 1 }, { 1, 0 } };
>   template<typename T>
>     constexpr pauli<T> sigma2 = { { 0, -1i }, { 1i, 0 } };
>   template<typename T>
>     constexpr pauli<T> sigma3 = { { 1, 0 }, { -1, 0 } };
> };
> ```
>
> —end example]

2. Modify paragraph 14/6 as follows:

> A function template, member function of a class template, variable template, or static data member of a class template shall be defined in every translation unit in which it is implicitly instantiated (14.7.1) unless the corresponding specialization is explicitly instantiated (14.7.2) in some translation unit; no diagnostic is required.

3. Modify paragraph 14.3.3/2

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

4. Modify paragraph 14.4/1

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

5. Add the following paragraph to section 14.7.1

> Unless a variable template specialization has been explicitly instantiated or explicitly specialized, the variable template specialization is implicitly instantiated when the specialization is used. A default template argument for a variable template is implicitly instantiated when the variable template is referenced in a context that requires the value of the default argument.

6. Modify 14.5.1.3/1:

> A definition for a static data member or static data member template may be provided in a namespace scope enclosing the definition of the static member's class template.

Add example:

> [Example:

```
struct limits {
   template<typename T>
      static const T min;     // declaration
};

template<typename T>
   const T limits::min = { }; // definition
```

—end example]

7. Modify paragraph 14.7.1/10 as follows:

   An implementation shall not implicitly instantiate a function template, a variable template, a member template, a non-virtual member function, a member class, or a static data member of a class template that does not require instantiation. [...]

8. Modify paragraph 14.7.1/11 as follows:

   Implicitly instantiated variable, class and function template specializations are placed in the namespace where the template is defined. [...]

9. Modify paragraph 14.7.2/1 as follows:

   A class, ~~a~~ function, variable or member template specialization can be explicitly instantiated from its template. [...]

10. Add the following to paragraph 14.7.2/3:

    If the explicit instantiation is for a variable or member function, the *unqualified-id* in the declaration shall be a *template-id*.

11. Modify paragraph 14.7.2/4:

    A declaration of a function template, a variable template, a member function or static data member of a class template, or a member function template of a class or class template shall precede an explicit instantiation of that entity. [...]

12. Modify paragraph 14.7.2/5:

[...] Otherwise, for an explicit instantiation definition the definition of a variable template, a function template, a member function template, or a member function or static data member of a class template shall be present in every translation unit in which it is explicitly instantiated.

13. Modify paragraph 14.7.2/6:

    An explicit instantiation of a class ~~or~~, function template, or variable template specialization is placed in the namespace in which the template is defined. [...]

14. Modify 14.7.2/10

    Except for inline functions, const variables of literal types, variables of reference types, and class template specializations, explicit instantiation declarations have the effect of suppressing the implicit instantiation of the entity to which they refer.

15. Add this bullet to paragraph 14.7.3/1

    — variable template

16. Modify paragraph 14.7.3/3

    A declaration of a variable template, a function template or class template being explicitly specialized shall precede the declaration of the explicit specialization. [...]

17. Modify paragraph 14.7.3/4

    A member function, a member function template, a member class, a member enumeration, a member class template, ~~or~~ a static data member, or a static data member template of a class template may be explicitly specialized for a class specialization that is implicitly instantiated; [...]

18. Modify paragraph 14.7.3/13:

An explicit specialization of a static data member of a template or an explicit specialization of a static data member template is a definition if the declaration includes an initializer; otherwise, it is a declaration.

19. Modify paragraph 14.7.3/17

    A specialization of a member function template ~~or~~, member class template, or static data member template of a non-specialized class template is itself a template.

20. Modify paragraph 7.1.5/1

    The `constexpr` specifier shall be applied only to the definition of a variable or variable template, the declaration of a function or function template, or the declaration of a static data member of a literal type (3.9). If any declaration of a function ~~or~~, function template, or variable template has constexpr specifier, then all its declarations shall contain the constexpr specifier.

4 Conclusion

This report proposes a simple extension to C++: allow 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.

5 Changes from N3615

These changes are based on straw polls from EWG meeting on 2013-04-17:

- A variable declared in a variable template need not be `const` or `constexpr`. A majority of EWG asked for the restriction to be removed even for the C++14 scope.
- A plurality of EWG felt that uses of variable templates as template template argument should be done separately, not for C++14.

Acknowledgement

Vicente J. Botet Escriba independently suggested support for constexpr variable templates on the `std-proposals` mailing list. Walter Brown's proposal [1] suggests the notion of "expression alias" modeled after the notion of template alias. Both proposals appear to overlap on key aspects regarding variable templates. This proposal, like its previous revision [2], does not propose that nullary function names automatically decay to a call.

References

[1] Walter Brown. Introducing Object Aliases. Technical report, ISO/IEC JTC1/SC22/WG21, 2013.

[2] Gabriel Dos Reis. Constexpr Variable Templates. Technical report, ISO/IEC JTC1/SC22/WG21, 2013.