Document Number: P0386R0
Date: 2016-05-30
Hal Finkel (hfinkel@anl.gov) and Richard Smith (richard@metafoo.co.uk)

Inline Variables

Proposal summary

The `inline` specifier can be applied to variables as well as to functions. A variable declared inline has the same semantics as a function declared inline: it can be defined, identically, in multiple translation units, must be defined in every translation unit in which it is odr-used, and the behavior of the program is as if there is exactly one variable.

Changes since N4424

N4424 was accepted by EWG with the following modifications:

- A namespace-scope inline variable must be of const-qualified (or reference) type.
- Initialization of inline variables is partially-ordered: if an inline variable is defined before some other variable in every translation unit in which the other variable is defined, it is initialized before that other variable.
- The `constexpr` specifier implies `inline` for variables as well as functions.

Proposed wording (relative to N4594)

In 3.1p2 change:

A declaration is a definition unless [...] it declares a non-inline static data member in a class definition (9.2, 9.4), it declares a static data member outside a class definition and the variable was defined within the class with the `constexpr` specifier (this usage is deprecated; see D.X), [...]

In 3.2p4 change:

Every program shall contain exactly one definition of every non-inline function or variable that is odr-used in that program; no diagnostic required. [...] An inline function or variable shall be defined in every translation unit in which it is odr-used.

In 3.2p6, change:

There can be more than one definition of a class type (Clause 9), enumeration type (7.2), inline function or variable with external linkage (7.1.2), [...] in a program provided that each definition appears in a different translation unit, and provided that the definitions [are the same]

In 3.5p3, change:

A name having namespace scope (3.3.6) has internal linkage if it is the name of

- [...]
- a non-inline variable of non-volatile const-qualified type that is neither explicitly declared extern nor previously declared to have external linkage; or
- [...]

In 3.6.3p1, change:

Dynamic initialization of a non-local variable with static storage duration is unordered if the variable is an implicitly or explicitly instantiated specialization, is partially-ordered if the variable is an inline variable that is not an implicitly or explicitly instantiated specialization, and otherwise is ordered [*Note*: an explicitly specialized non-inline static data member or variable template specialization has ordered initialization. — *end note*].<paragraph break>

Dynamic initialization of non-local variables V and W with static storage duration are ordered as follows:

- If V and W have ~~Variables with~~ ordered initialization and V is defined before W within a single translation unit, ~~they shall be initialized in the order of their definitions in the translation unit~~ the initialization of V is sequenced before the initialization of W.
- If V has partially-ordered initialization and W does not have unordered initialization and V is defined before W in every translation unit in which W is defined, the initialization of V is sequenced before the initialization of W if the program does not start a thread (30.3) and otherwise happens before the initialization of W.
- Otherwise, if a program starts a thread ~~(30.3)~~ before either V or W is initialized, the ~~subsequent~~ initializations of V and W are ~~a variable is~~ unsequenced~~ with respect to the initialization of a variable defined in a different translation unit~~.
- Otherwise, the initializations of V and W are ~~a variable is~~ indeterminately sequenced~~ with respect to the initialization of a variable defined in a different translation unit~~. ~~If a program starts a thread, the subsequent unordered initialization of a variable is unsequenced with respect to every other dynamic initialization. Otherwise, the unordered initialization of a variable is indeterminately sequenced with respect to every other dynamic initialization.~~

[*Note:* This definition permits initialization of a sequence of ordered variables concurrently with another sequence. — *end note*]

In 3.6.3p2, change:

It is implementation-defined whether the dynamic initialization of a non-local non-inline variable with static storage duration happens before the first statement of `main`. If the initialization is deferred to happen after the first statement of `main`, it happens before the first odr-use (3.2) of

any non-inline function or variable defined in the same translation unit as the variable to be initialized. [...]

Add a new paragraph after 3.6.3p2:

It is implementation-defined whether the dynamic initialization of a non-local inline variable with static storage duration happens before the first statement of main. If the initialization is deferred to happen after the first statement of main, it happens before the first odr-use (3.2) of that variable.

In 3.6.3p3, change:

It is implementation-defined whether the dynamic initialization of a non-local non-inline variable with static or thread storage duration is sequenced before the first statement of the initial function of the thread. If the initialization is deferred to some point in time sequenced after the first statement of the initial function of the thread, it is sequenced before the first odr-use (3.2) of any non-inline variable with thread storage duration defined in the same translation unit as the variable to be initialized.

In 7.1p1, add inline to the list of decl-specifier. Add a new subclause, "The inline specifier", 7.1.X, as follows:

p1: The `inline` specifier can be applied only to the declaration or definition of a variable or function.

Move 7.1.2p2 ("A function declaration with an inline specifier declares an inline function [...]") unchanged into 7.1.X as p2

p3: A variable declaration with an `inline` specifier declares an *inline variable*. A namespace-scope inline variable shall be either of const-qualified type or of reference type.

Move 7.1.2p3 into 7.1.X, split into two paragraphs, and modify as indicated:

p4: A function defined within a class definition is an inline function.<paragraph break>

p5: The `inline` specifier shall not appear on a block scope ~~function~~ declaration.[*Footnote*] If the `inline` specifier is used in a friend function declaration, that declaration shall be a definition or the function shall have previously been declared inline.

Move 7.1.2p4 into 7.1.X and modify as indicated:

p6: An inline function or variable shall be defined in every translation unit in which it is odr-used and shall have exactly the same definition in every case (3.2). [*Note*: A call to the inline function

or a use of the inline variable may be encountered before its definition appears in the translation unit. — *end note*] If the definition of a function or variable appears in a translation unit before its first declaration as inline, the program is ill-formed. If a function or variable with external linkage is declared inline in one translation unit, it shall be declared inline in all translation units in which it appears; no diagnostic is required. An inline function or variable with external linkage shall have the same address in all translation units. [*Note:* A static local variable in an extern inline function always refers to the same object (3.2). A type defined within the body of an extern inline function is the same type in every translation unit. — *end note*]

Remove `inline` *from the list of function specifiers in 7.1.2p1.*

In 7.1.5p1, change:

The `constexpr` specifier shall be applied only to the definition of a variable or variable template~~,~~ or the declaration of a function or function template~~, or the declaration of a static data member of a literal type (3.9)~~. A function or variable declared with the `constexpr` specifier is implicitly an inline function or variable (7.1.X). If any declaration of a function or function template has a `constexpr` specifier, then all its declarations shall contain the `constexpr` specifier. [...]

In 7.1.5p2, change:

A `constexpr` specifier used in the declaration of a function that is not a constructor declares that function to be a *constexpr function*. Similarly, a `constexpr` specifier used in a constructor declaration declares that constructor to be a *constexpr constructor*. ~~`constexpr` functions and `constexpr` constructors are implicitly inline functions (7.1.2).~~

In 9.2.3.2p2, change:

The declaration of a non-inline or uninitialized static data member in its class definition is not a definition and may be of an incomplete type other than cv-qualified void.

In 9.2.3.2p3, change:

If a non-volatile non-inline const static data member is of integral or enumeration type, its declaration in the class definition can specify a *brace-or-equal-initializer* in which every *initializer-clause* that is an *assignment-expression* is a constant expression (5.20). ~~A static data member of literal type can be declared in the class definition with the constexpr specifier; if so, its declaration shall specify a brace-or-equal-initializer in which every initializer-clause that is an assignment-expression is a constant expression. [Note: In both these cases, the member may appear in constant expressions. — end note]~~ The member shall still be defined in a namespace scope if it is odr-used (3.2) in the program and the namespace scope definition shall not contain an initializer. An inline static data member can be defined in the class definition and may specify

a *brace-or-equal-initializer*. If the member is declared with the constexpr specifier, it may be redeclared in namespace scope with no initializer (this usage is deprecated; see D.X). Declarations of other static data members shall not specify a *brace-or-equal-initializer*.

In 14.7.2p10, change:

Except for inline functions and variables, declarations with types deduced from their initializer or return value (7.1.6.4), 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. [...]

In 14.7.3p12, change:

An explicit specialization of a function or variable template is inline only if it is declared with the `inline` specifier or defined as deleted, and independently of whether its function or variable template is inline. [*Example*:

```
template<class T> void f(T) { /* ... */ }
template<class T> inline T g(T) { /* ... */ }
template<> inline void f<>(int) { /* ... */ } // OK: inline
template<> int g<>(int) { /* ... */ } // OK: not inline
```

— *end example*]

In Annex D, add a new subclause, "Redeclaration of static constexpr data members", D.X, with the following content:

For compatibility with prior C++ International Standards, a constexpr static data member may be redundantly redeclared outside the class with no initializer. This usage is deprecated. [Example:

```
struct A {
  static constexpr int n = 5; // definition (declaration in C++ 2014)
};
const int A::n; // redundant declaration (definition in C++ 2014)
```

— end example]