Document Number: N4424
Date: 2015-04-07
Hal Finkel (hfinkel@anl.gov) and Richard Smith (richard@metafoo.co.uk)

Inline Variables

Introduction

C++ generally requires all extern functions and variables to have exactly one definition in the program, and has long provided a facility to relax this requirement for functions via the inline specifier. inline functions can be defined in multiple translation units (and must be defined in all translation units in which they are odr-used), but all such definitions must be the same. The same is true (through a different mechanism) for functions and variables instantiated from templates. No such facility exists for non-template variables, however, and as a result, all non-template static class data members need to be explicitly defined in some translation unit. This is true even if all other definitions associated with the class can reside in header files that make minimal use of the preprocessor.

However, it is not uncommon to desire the existence of a globally-unique object without having to pick a single translation unit in which to define it. As a practical matter, making this choice generally requires either the use of non-trivial preprocessor macros, separately compiled libraries, or both. However, one strength of C++ is its ability to support the development of header-only libraries. In this vein, the lack of the ability to define an inline variable poses a significant constraint on library design.

Three partial workarounds currently exist. The first is to use an inline getter function that defines a static local variable. 7.1.2p4 already requires that "A static local variable in an extern inline function always refers to the same object."[1] Thus, a getter function that returns a reference to a static local variable declared within it can provide a multiply-defined global object. Drawbacks to this approach include: (a) The introduction of otherwise-unnecessary functions (and associated function-call operators) and (b) The object won't be constructed until the getter function is first called (6.7p4) (which rules out some use cases discussed below).

The second workaround is to use a static data member of a class template or a variable template, as definitions of these are allowed to appear in multiple translation units (3.2p6). Drawbacks to this approach include: (a) The introduction of otherwise-unnecessary template parameters and (b) The requirement of a use of the object to force its existence (14.7.1p2) (which rules out some use cases discussed below).

[1] This is also implied by 3.2p6, for an inline function definition D appearing in multiple translation units, “the behavior is as if there were a single definition of D.”

The third workaround is to use a lifetime-extended temporary bound to a static reference data member of a class:

```
struct A {
  constexpr static auto &&kFoo = createObject();
};
```

This works because it is not possible to odr-use a reference that is initialized by a constant expression so no definition of `kFoo` is required, and an implementation is required to act as if there was only one definition of the surrounding class so the program contains only a single temporary object. However: (a) it requires that the initializer of the object is a constant expression, (b) it is an extremely obscure technique that is hard to understand even once explained, and (c) it does not work in practice on many C++ implementations.

The currently-available workarounds cannot be used with objects that have useful side effects simply as a result of being created (process-monitoring objects, for example). Such an object fulfills its role simply by being constructed, and must not need to be explicitly used elsewhere. The workaround of using a static local variable in an extern inline function does not work for this because a caller of the function is required to trigger the construction. The workaround of using a templated variable also does not work because, as 14.7.1p2 says, "the initialization (and any associated side-effects) of a static data member does not occur unless the static data member is itself used in a way that requires the definition of the static data member to exist" (and likewise for a variable template specialization). One can, of course, explicitly instantiate the variable, but such an instantiation would itself be a definition (because it would need to include an initializer) (14.7.3p13), and could only appear in one translation unit. The workaround of using a lifetime-extended temporary bound to a static reference data member of a class does not work because the object initializer will not be a constant expression.

Relationship to Prior Proposals

The work was motivated by the presentation of N4147 at Urbana; this proposal is not based on N4147, and is substantially simpler with a more limited scope.

Implementation Concerns

All conforming implementations likely support the basic infrastructure necessary to implement this proposal; it is very similar to what is needed to support static local variables defined within extern inline functions and static data members of class templates.

Proposed Changes

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), [...]

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 3.6.2p2, change:

Dynamic initialization of a non-local variable with static storage duration is unordered if the variable is an implicitly or explicitly instantiated specialization, or is an inline variable, and otherwise is ordered.

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

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

p2: A variable declaration with an `inline` specifier declares an *inline variable*.

Move 7.1.2p2 unchanged into 7.1.X as p3.
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 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 9.4p2, 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 9.4p3, change:

If a non-volatile 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 not inline and is odr-used (3.2) in the program and the namespace scope definition shall not contain an initializer. Except as specified above, a non-inline declaration of a static data member shall not specify a *brace-or-equal-initializer*.

Examples

A simple static data member:

```
struct WithStaticDataMember {
  // This is a definition, no out-of-line definition is required.
  static inline constexpr const char *kFoo = "foo bar";
};

template<typename T> struct X {
  // This does not require an out-of-line definition either.
  static inline int x;
};
```

A monitor object:

```
class Monitor {
public:
  Monitor() {
    // Start a thread, call std::set_new_handler, etc.
  }
};

// This can be declared in a common header file included in
// multiple translation units, and we’ll get one TheMonitor
// object in the program.
inline Monitor TheMonitor;
```

Avoiding ODR violation with global constants:

```
// This object’s address is captured by an inline function,
// and so must be extern, and inline is helpful here to avoid
// needing to pick a translation unit for the definition.
inline extern const n = 5;

// This inline function takes the address of n, and so to avoid ODR
violation, n must not have internal linkage.
inline auto naddr() { return &n; }
```

A class with a constexpr static data member of its own type:

```
struct Foo {
  static const Foo kNull; // cannot be defined here,
                          // Foo is not yet complete
  int a, b, c;
};
inline constexpr Foo::kNull = {};
```

Note that kNull must be defined in the header in order to be usable in constant expressions.