Document #:D3334R0

Date: 2024-06-16

Project: Programming Language C++

Audience: SG7

Evolution Work Group Incubator

Reply-to:

Coral Kashri <coralkashri@gmail.com>

Cross Static Variables

I. Introduction

“Static” variables declared within template functions or class contexts have existed since templates were introduced to C++. We propose extended behavior: a variable that can be used within a template context (template function, class), without being dependent on a specific instantiation.

II. Motivation and Scope

Today, if one wants to do cross-template calculations, for example, counting the number of times a certain template function gets called without separating to instantiations, they'll need to use one of the following methods:

1. Using the global scope, and possibly having to deal with translation unit issues.
2. Using “magic singleton” in a way that will be discussed here further.
3. Side note: In the case of template classes, or template functions in classes, there is another solution of driving from a common non-template class. This proposal doesn’t focus on these cases, even though it’ll also solve the functions’ case here.

This proposal gives the ability to do that for both template functions and template classes without having to employ any of these methods.

Common use cases could be:

- Allocators - the same memory map pool could be used across multiple instantiations.
- Shared resources - e.g., an output stream that can be used as a logger.
- Shared statistics - e.g., call counts across multiple instantiations (might be useful for debugging sessions).

All of these will be possible without the need for global variables while restricting the usage to only the instantiated functions.

III. Impact On the Standard

We are proposing a language-only strict addition. No existing well-formed code will be affected in any way (assuming we make syntactic choices which do not conflict with existing names in current codebases).

IV. Design Decisions

Syntactic Possibilities for Introducing the Variable

1. Introduce a boolean param to “static” keyword

Adding the option to call *static* keyword with a boolean parameter. Static(false) or just static without parameters will keep the current behavior - fully dependent on the instantiation. Static(true) would mean that the variable can be accessed and changed from any instantiation.

Example:

```
template <int Num>
void func() { static(true) int n = 5; std::cout << n++; }

func<1>();
func<2>();

Would print: 56

Example 2:
template <int Num>
struct A {
    static(true) int ci = 0;
    static/*(false)*/ int ri = 0;
    A() { ++ci; ++ri; }
```

```
};

A<1> a1;
A<2> a2;

assert(a1.ci == 2);
assert(a1.ri == 1);
assert(a2.ci == 2);
assert(a2.ri == 1);
assert(&a1.ci == &a2.ci);
assert(&a1.ri != &a2.ri);
```

Constraints:
The boolean argument of the static must be a non-dependent core constant expression.

Cons:

- The boolean might be confusing. It's unclear whether true is a cross-static variable or not.
- The boolean is confusing as a dependent expression is not allowed, making it probably the only such case in the current standard. For example, this rule doesn't apply to the *explicit* keyword.

Note: Instead of a boolean argument, it’s possible to use an enum value resulting in clearer code.

2. Add a new keyword which can be appended before the “static” keyword

Adding a new keyword, in this paper named “cross” (bikeshedding is welcome). The keyword can be used before the “static” keyword. To get the new behavior, add this keyword before the static.

For example:

```
template <int Num>
void func() { cross static int n = 5; std::cout << n++; }

func<1>();
func<2>();
```

Would print: 56

Cons:

- Adding a new keyword is usually frowned upon and may delay approval by WG21, possibly missing the C++26 deadline.
- Agreeing on such a keyword that is unlikely to appear in existing codebases may be tricky.

Pros:

- Clear meaning what it means (assuming we choose the new keyword wisely).
- The keyword clearly appertains to the static variable declaration, and it's more obvious that it can't be dependent on the template arguments.

Consider the example:

```
template <int Num>
void func() {
     if constexpr(...)
          cross static int n = 5; std::cout << n++;
     else
          static int n = 5; std::cout << n++;
}
```

3. A new keyword “cross_static”

Similar to the previously discussed approach, just without a possibly separated context for “cross” keyword.

Pros:

- Only has a known purpose described in this draft.
- A keyword made up from two different words is more likely than a single-word keyword to not break any existing codebases.

Cons:

- If in the future there will be another proposal discussing another possible usage of “cross” keyword, there will be a weird mix of “cross_static” and “cross something_else”.

4. Double static keyword usage

Using the “static” keyword twice in a row, to indicate that this is a “higher level” static than the usual one.

For example:

```
template <int Num>
void func() { static static int n = 5; std::cout << n++; }

func<1>();
```

```
func<2>();
```

Would print: 56

Pros:

- No new keyword requirement.

Cons:

- Less clear than previously discussed options. Will be hard to teach new students & old standards users.
- Easier to make mistakes when the same keyword is used twice, with that kind of potential impact on the results.

5. An entirely new keyword, without using “static”

Should be a keyword that doesn’t convey any conflicting meaning and which doesn’t already have some other use. Should be a single word. Possible candidates:

- persistent
- inherent

Pros:

- Easy to parse.

Cons:

- A strong link to “static” isn’t immediately obvious, intuition would require time to build up.
- Existing codebases might already be using this word in different contexts. As such and as a single word, it may be harder for the compiler to disambiguate it as a contextual keyword.
- The compiler must actively detect any use of this word + static together, and make such code ill-formed.

6. A new attribute appertaining to “static”

For example:

```
template <int Num>
void func() { [[cross]] static int n = 5; std::cout << n++; }

func<1>();
func<2>();
```

Would print: 56

Pros:

- Easy to parse, and shouldn't have any ambiguities.
- Attribute context allows even keywords currently in use elsewhere to not create conflicts (unless they are used as attributes elsewhere).

Cons:

- The standard defines attributes as ignorable. On one hand, this allows older compilers to parse such code, but in our case, this would give the code a different meaning.

Other Design Considerations

- A cross-static variable *value* should not be dependent on the template arguments.
- The type of a cross-static variable should not depend on the template arguments (like std::condition_t, typename, etc..).

Request for polls

The author would like feedback from SG7/EWGI on the preferred syntax (between options 1-6).

V. Technical Specifications

A cross-static variable should:

1. Have a single unique address between:
    a. All template function instantiations.
    b. Different translation units.
2. Be easy to use & understand.
3. Have thread-safe lazy initialization like a regular static variable. (Being lazily initialized automatically shields it from the "static initialization order fiasco".)

VI. Implementation

A syntactic sugar for "magic singleton". Using a static variable in an external non-dependent function, we can achieve all of the abilities discussed above.
Whenever a cross-static variable is defined, the compiler will generate a hidden function that contains the definition of this static variable, lazily initializes it (via "magic statics"), and provides access to it by reference. The hidden function will be implicitly invoked in every place in the source

function where program flow may encounter the cross-static variable for the first time. The hidden function will have template linkage semantics, requiring the linker to discard all but one of its compiled instances, which will be shared by all the TUs.

Godbolt examples:

- A short example of different translation units usage: https://godbolt.org/z/9oW383PxP
- Full example including template instantiations in different translation units: https://godbolt.org/z/j4xM6PGh6

VII. Acknowledgements

I would like to thank Andrei Zissu, Tal Yaakovi, and Inbal Levi for early feedback on this paper.