_Thread_local for better C++ interoperability with C

Document number: ISO/IEC JTC 1/SC 22/WG 14 N 2850
ISO/IEC JTC 1/SC 22/WG 22 P2478R0
Date: 2021-10-14
Audience subgroups: WG 14, WG 21 SG 22, EWGI
Revises: None
Reply-to: Hubert S.K. Tong <hubert.reinterpretcast@gmail.com>
Rajan Bhakta <rbhakta@us.ibm.com>

Motivation

C and C++ both provide facilities for “thread-local” variables. In C17, _Thread_local is a keyword, used (as a storage-class specifier) to declare variables with thread storage duration. The corresponding keyword in C++ is thread_local. However, superficial similarities, including the availability of a thread_local macro in C17’s <thread.h> somewhat mask a more fundamental difference between the two languages: that C++ supports dynamic initialization and also types requiring destruction. This difference has practical effects, since a common implementation strategy requires additional complexity in accessing C++ thread_local variables in various contexts compared to accessing C _Thread_local variables. This paper identifies various possible directions for acknowledging this difference for use in source code that is meant to be consumed by both C and C++.

Summary of Changes

- Original proposal

Background and Introduction

In C++, variables with thread storage duration may have dynamic initialization or non-trivial destruction. Such initialization may occur for non-block variables at thread startup or be deferred up to the first non-initialization use. An implementation strategy used by various compilers, including GCC and Clang, is to perform deferred initialization for non-block variables with thread storage duration by inserting a function call where the variable is used (depending on context). The function call causes any necessary initialization to occur. The function call itself and the function definition introduces cost in the form of runtime and code size overhead: a cost that can be avoided if the variable is known to have neither dynamic initialization nor non-trivial destruction. C++20 adds the constinit keyword as a method to inform the compiler that a variable should not have dynamic initialization. There is no similar mechanism in C++20 to indicate that an incomplete class type has non-trivial destruction.

Beyond a “mere” overhead cost, it is also worth noting that, in some environments, variables defined _Thread_local in C do not provide all of the auxiliary symbols necessary for its usage as a C++ thread_local variable; therefore, references to the C-defined variable from C++ potentially result in link errors.

It is also the case that, in practice, accessing non-block variables of static storage duration defined in C++ with dynamic initialization or non-trivial destruction from C code is not really more dangerous than accessing the same from C++ code: major implementations perform the initialization at program startup. The same cannot be said for trying to access a similar variable with thread (instead of static) storage duration: the access from C has a good chance of observing the variable prior to initialization.

Proposal for C2X

It is observed that WG 14 document N2654, “Revise spelling of keywords v5”, proposes to replace _Thread_local with thread_local as the preferred form in the C standard. Given the information presented in this paper, the C committee may want to hold back on taking a direction which advocates increased usage of thread_local as opposed to _Thread_local. This paper proposes to maintain the status quo of C17 where thread_local is a macro, keeping _Thread_local, with the known differences from C++, the preferred form.

Future Directions

For C++, it is noted that unnecessary cost occurs in practice even with constinit thread_local where incomplete class types are involved. There may be a desire for a stronger constinit or to leverage P1247’s no_destroy attribute (or a non-attribute version of the same) to solve this issue in a more general manner; however, _Thread_local already exists in C, is implemented as an extension in C++ by Clang, and can be used to convey the intended code generation semantics. As an additional data point: at the time of this writing, neither of Clang’s [[clang::no_destroy]] nor -fno-c++-static-destructors work for this purpose. GCC and Clang both offer __thread in both C and C++ mode (with the intended code generation semantics). It is noted that implementations currently do not diagnose cases where non-defining declarations of variables with complete class types having manifest non-trivial initialization are declared with __thread; however, diagnostics are emitted when definitions of such variables are found to have dynamic initialization or if its type (i.e., ignoring no_destroy) has non-trivial destruction.

For C, it may be worth exploring the addition of constinit as a keyword with enforcement of the constraints associated with its use in C++. Users can then declare their variables constinit consistently between C and C++. It is also possible to consider having constinit be a macro in C that expands to no tokens; however, that C++ keyword is only tangentially related to thread storage duration–making the choice of which header to place the macro in more difficult. If C++ eventually adopts _Thread_local, then deprecating the thread_local macro may also be appropriate.

Acknowledgements

The authors would like to thank Aaron Ballman, the chair of the Study Group for C and C++ compatibility, for giving encouragement for the preparation of this paper and providing assistance with scheduling its discussion.

Bibliography

Gustedt. ISO/IEC JTC 1/SC 22/WG 14 N 2654, “Revise spelling of keywords v5”. http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2654.pdf

Lopes, Bastien, and Pilkington. ISO/IEC JTC 1/SC 22/WG 22 P1247R0, “Disabling static destructors: introducing no_destroy and always_destroy attributes”. https://wg21.link/p1247r0