Destructor Semantics Do Not Affect Constructible Traits

Addressing LWG#2116 and LWG#2827

Document #: P2842R0
Date: 2023-05-11
Project: Programming Language C++
Audience: Library Working Group
Revises: N/A

Reply-to: Alisdair Meredith
<ameredith1@bloomberg.net>
Harold Bott Jr.
<hbott1@bloomberg.net>

Contents

1 Abstract

For type traits that deal with constructors, the Standard's wording was carefully drafted to avoid invoking destructors in determining the `::value` of the trait, as would occur if querying an expression using a temporary. LWG issue 2116 raised concerns that this formulation still required an accessible destructor, and subsequently implementations treated the specification as-if that destructor were indeed part of the query's semantics.

This paper presents the history and motivation for the wording and the competing history of the implementations and provides alternative resolutions to make the Standard and implementations agree.

2 Revision History

2.1 R0: Varna 2023

Initial draft of the paper.

3 Introduction

Type traits were introduced to C++ in 2003 by [N1424] for the Library Extensions TR1 [N1836] and were subsequently added to the C++11 draft when the Library Extensions TR1 was merged into the C++ working draft at the 2006 Berlin meeting.

NB comments addressed by [N3142] changed the clear wording that considered only constructors for these traits, to consolidated wording relying on an invented variable declaration that was intended to have the same meaning.

Following [N3142], the interpretation adopted by the Standard Library vendors (with regard to the destructors) was the opposite of that originally intended meaning. Vendors treat the invented variable *declaration* as if it were an expression rather than a statement.

Meanwhile, [LWG2116] raised concerns about the `is_nothrow_constructible` trait requiring an *accessible* destructor; subsequently, [LWG2827] raised similar concerns for the `is_trivially_*_constructible` traits.

This paper details a number of ways to resolve those two issues, correcting the Standard's wording, the Library implementations, or both.

4 Wording History

To understand the issues at hand, we will look into the history of how the current specification evolved.

4.1 Specified in terms of the constructor alone

Type traits were originally introduced in [N1836] where the predicate for a trait named `has_trivial_copy` is specified as

> The copy constructor for `T` is trivial

Similarly, the predicate for a trait named `has_nothrow_copy` is specified as

> The copy constructor for `T` has an empty exception specification or
> can otherwise be deduced never to throw an exception

Note that in C++03, an empty exception specification refers to `throw()`, not to the absence of any exception specification, i.e., equivalent to a `noexcept(true)` exception specification today.

These traits were renamed to `has_trivial_copy_constructor` and `has_nothrow_copy_constructor` to clarify the intent in their name as part of the subsequent C++0x work and would be renamed again before C++11 was complete.

4.2 Attempts to implement the `noexcept` traits purely in the library

The original specification for `has_nothrow_copy_constructor` required a compiler intrinsic to implement, and it was thought that when we added `noexcept` to the language in [N3050], we should be able to provide a pure library solution with something like

```
has_nothrow_copy_constructor<T>::value = noexcept(T{declval<T const &>()});
```

The problem with this formulation is that the expression inside the `noexcept` operator includes the destruction of the temporary `T` object. At that time, destructors without an exception specification were implicitly `noexcept(false)`, with work in EWG to independently address this `noexcept` behavior as part of a larger issue in the design of the language.

4.3 Renamed to final form

A variety of NB comments were filed against these traits for the C++11 FCD, and all were resolved in the omnibus issue paper [N3142]. The first part of that was to rename the `has_*_constructor` traits to `is_*_constructible` traits.

4.4 Specified in terms of an invented variable declaration

In addition to changing the names of traits, the omnibus issue paper [N3142] changed the way `constructible` traits were specified, i.e., consistently in terms of the `is_constructible` trait.

Quoting the rationale for this change to the `is_nothrow_constructible` trait:

> `is_constructible<T, Args...>::value` is true and the variable definition for `is_constructible`, as defined below, is known not to throw any exceptions (5.3.7 [expr.unary.noexcept]).
>
> [Rationale: We rely on the core language meaning here and therefore suggest to add a reference to 5.3.7 [expr.unary.noexcept], because this allows us to implicitly point to the noexcept operator as a tool to recognize this property in situations where expressions are involved. Some type traits are defined in terms of variable definitions and we therefore cannot directly use the operator for them. A correct implementation will be based on a compiler-intrinsic here, simulating implementations will typically partition into different expressions, each of them could be checked via noexcept.
>
> Referring to 5.3.7 [expr.unary.noexcept] has the advantage that this sub-clause directly refers to 15.1 [except.throw] where we find
>
>> Throwing an exception transfers control to a handler.
>
> In the context we use the term it is clear that this means that the final handler is not located within the expression or definition that is the result of an exception.]

You will note that none of this rationale refers to execution of destructors; however, the well-formed constraints and checking for side effects would include checking for the accessibility of the destructor. (This is the origin of LWG issue 2116.) The interpretation of

> is known not to throw any exceptions (5.3.7 [expr.unary.noexcept])

is explicitly delegated to the core language describing how the `noexcept` operator is required to make the same determination.

The hidden issue is the interpretation of

… the following variable definition would be well-formed for some invented variable `t`:

```
        T t(create<Args>()...);
```

The phrasing here is chosen very deliberately to exclude the destructor that is executed at the end of the lifetime of `t`; otherwise, we would have more clearly written this same condition as

… the following statement would be well-formed for some invented variable `t`:

```
        { T t(create<Args>()...); }
```

That the variable declaration ends at the `;` and that the destructor does not run until after the `}` that ends the block scope were well known at the time; that is why such a scope is deliberately absent in the invented variable form.

If the current interpretation that includes the call of the destructor were the intended meaning, we would not need to call into some invented language requiring a compiler intrinsic (as observed in the rationale) to decouple the lifetime from the construction. We could simply write an expression that creates a temporary using the appropriate constructor, and the execution of the destructor is then implied:

~~`is_constructible<T, Args...>::value` is `true` and the variable definition for `is_constructible`, as defined below, is known not to throw any exceptions (5.3.7[expr.unary.noexcept]).~~

`noexcept(T(declval<Args>)...)` is `true`

4.5 Wording applied to trivial constructor traits too

Once the specification for `is_nothrow_move_constructible` was rephrased in terms of an invented variable (through the dependency on the `is_constructible` trait), the same formulation was applied to `is_trivially_copy_constructible`. Given the original intent of the "invented variable declaration" wording, this change should not have impacted implementations, which would still require the same compiler intrinsic as `has_trivial_copy_constructor` to obtain information about the triviality of copy constructors.

5 History of Library Behavior

The following program was tested with Godbolt Compiler Explorer against a range of Standard Libraries, building in C++11 mode (or later where there are no earlier versions available):

```
#include <type_traits>

struct Test {
   Test() = default;
   Test(Test const&) = default;
  ~Test() noexcept(false) {}  // non-trivial, potentially throwing
};

static_assert(std::is_trivially_copy_constructible<Test>::value, "non-trivial");
static_assert(std::is_nothrow_copy_constructible<Test>::value, "may throw");
```

To the best of the authors' knowledge, all libraries that support the traits by the `is_*_constructible` name fail both of the static assertions. If the exception specification is removed from the destructor, then the `is_nothrow_copy_constructible` assertion passes. If the destructor is defaulted or removed, then the `is_trivially_copy_constructible` assertion passes as well.

Older GCC compilers shipped with implementations of the earlier traits too, so they were tested with the following program, which would be equivalent to the one above even on compilers too old for experimental support for `noexcept`:

```
#include <type_traits>

struct Test {
   Test() = default;
   Test(Test const&) = default;
  ~Test() throw(int) {}  // non-trivial, potentially throwing
};

static_assert(std::has_trivial_copy_constructor<Test>::value, "non-trivial");
static_assert(std::has_nothrow_copy_constructor<Test>::value, "may throw");
```

This program passes both assertions with GCC compilers from 4.4 to 4.6, after which the `has_nothrow_copy_constructor` trait was removed.

Finally, the hybrid program below was tested with later versions up to GCC 6.4 (after which the pre-Standard `has_trivial_copy_constructor` trait was removed):

```
#include <type_traits>

struct Test {
   Test() = default;
   Test(Test const&) = default;
  ~Test() noexcept(false) {}  // non-trivial, potentially throwing
};

static_assert(std::has_trivial_copy_constructor<Test>::value, "non-trivial");
static_assert(std::is_trivially_copy_constructible<Test>::value, "non-trivial");
static_assert(std::is_nothrow_copy_constructible<Test>::value, "may throw");
```

Here, the `is_*_constructible` traits consistently fail the assertions, as in our first test, and the `has_trivial_copy_constructor` assertion continues to pass.

Hence, we conclude that the `is_*_constructible` traits have always been implemented with the semantics they have today, which was clearly a change from the implementation of the preceding `has_*_constructor` traits.

6 Resolving the Issues

Due to the disconnect between the intended meaning of the current specification and the current implementations of that specification, we need to decide which to prefer, and amend the Standard to clarify accordingly.

6.1 Motivation to retain original intent

The traits, as specified with their original intent and naming, offer a type query users cannot write themselves, since it is not possible to decouple constructor calls from *all* other language features in a SFINAE-able expression. That is, invoking a constructor is not possible without implicitly invoking at least one more expression; without the "if a declaration were well-formed" clause, we would have to choose between an expression involving a temporary (and so invoking the destructor) or an expression involving `new` (which is an overloadable operator with a larger set of failure modes).

When we consider the current implementations of the `is_trivially_*_constructible` traits, they become essentially redundant since the difference between `is_trivially_copyable` and `is_trivially_copy_constructible` is negligible; the missing constraint for the predicate for `is_trivially_copy_constructible` is that the copy assignment operator, and any eligible move constructor or move assignment operator, are also trivial. Observe that trivially copy constructible types are not necessarily trivially copyable types, as such types can violate the missing constraint.

Requiring a destructor to be accessible and have the same queried property as the selected constructor turns these traits into compound traits with a confusing name; since there is no intuitive connection between `is_constructible` and `is_destructible`, they are independent behaviors.

6.2 Motivation to retain the current semantics

The semantics that all current libraries provide (at the time this paper is being written) assume the object ends its lifetime at the end of the declaration. This has always been the case under the current name, so it might be too late to change the implementations.

The main concern with this approach is that the `trivial` traits serve no purpose. They do not guarantee a type is trivially copyable, enabling some bit-manipulating optimizations. On the other hand, they no longer provide reflection on the queried operation.

We see two simple ways to address these concerns.

1 We can deprecate the `is_trivially_*` traits other than the trait that maps to language support, `is_trivially_copyable`.

2 Rather than requiring a trivial destructor, we require a trivially copyable type, which would enable the bitwise manipulations *and* ensure that the appropriate call is satisfied by the public interface of that type.

Of these two directions, we prefer the second.

In any event, a note should be added to the Standard to clarify the intent that the destructor semantics are intended as part of the invented variable declaration.

7 Proposed Resolution (NAD)

The proposed resolution is to close these issues as NAD, indicating that bug reports should be filed against the existing library implementations, contrary to the rationale currently recorded in the issues.

7.1 Close the issue as NAD

Add the following rationale to the issue(s):

Varna, 2023:

The trait correctly relies on an accessible destructor, which is strictly correct according to the language. While an entity may be initialized with dynamic storage duration without an accessible destructor and allowed to leak, such use is possible only through the use of the `new` operator, which opens up a variety of additional overloads that might affect the result and so should be considered as a different trait.

7.2 Clarify the Standard so the interpretation is clear

21.3.5.4 [meta.unary.prop] Type properties

9 The predicate condition for a template specialization `is_constructible<T, Args...>` shall be satisfied if and only if the following variable definition would be well-formed for some invented variable `t`:

```
        T t(declval<Args>()...);
```

[*Note 7:* These tokens are never interpreted as a function declaration. —*end note*]

Access checking is performed as if in a context unrelated to `T` and any of the `Args`. Only the validity of the immediate context of the variable initialization is considered.

[*Note X:* The accessibility of the destructor is checked as part of a variable declaration but the destructor is not called until later, so no other aspects such as whether the destructor is trivial or `noexcept` are relevant. *—end note*]

[*Note 8:* The evaluation of the initialization can result in side effects such as instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. *—end note*]

8 Alternate Resolution

If we were to canonize the existing library implementations' behavior, then the wording should be updated as follows, relative to [N4944].

8.1 Option A

Change the invented variable declaration into a temporary expression, which also removes any ambiguity of the declaration being interpreted as a function declaration by a vexing parse.

8.1.1 Proposed changes

21.3.5.4 [meta.unary.prop] Type properties

9 The predicate condition for a template specialization `is_constructible<T, Args...>` shall be satisfied if and only if the ~~following variable definition~~ expression `T(declval<Args>()...)` would be well-formed ~~for some invented variable t:~~.

```
~~T t(declval<Args>()...);~~
```

~~[*Note 7:* These tokens are never interpreted as a function declaration. *—end note*]~~

Access checking is performed as if in a context unrelated to `T` and any of the `Args`. Only the validity of the immediate context of the ~~variable initialization~~ expression is considered.

[*Note 8:* The evaluation of the ~~initialization~~ expression can result in side effects such as the destruction of the temporary, instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. *—end note*]

Editor's note: "temporary" rather than "temporary object" is correct, to allow for the initialization of references.

8.2 Option B

Change the invented variable declaration into a temporary expression, which also removes any ambiguity of the declaration being interpreted as a function declaration by a vexing parse; move the wording directly into the type traits table.

8.2.1 Proposed changes

Take the wording of Option A and place it directly into the type traits tables, as there is no need for special wording describing an invented variable any more.

8.3 Option C

Introduce braces around the invented variable declaration for the minimal change that makes the "includes destructor" interpretation clear:

8.3.1 Proposed changes

21.3.5.4 [meta.unary.prop] Type properties

9 The predicate condition for a template specialization `is_constructible<T, Args...>` shall be satisfied if and only if the following variable definition would be well-formed for some invented variable `t`:

```
{ T t(declval<Args>()...); }
```

[*Note 7:* These tokens are never interpreted as a function declaration. —*end note*]

Access checking is performed as if in a context unrelated to `T` and any of the `Args`. Only the validity of the immediate context of the variable initialization is considered.

[*Note 8:* The evaluation of the initialization can result in side effects such as instantiation of class template specializations and function template specializations, the generation of implicitly-defined functions, and so on. Such side effects are not in the "immediate context" and can result in the program being ill-formed. —*end note*]

[*Note 9:* The block scope makes it clear that the destructor is part of the semantic, and the reference to initialization means the requirements are tested only against the declaration of the destructor for type `T`, which is enough to verify its triviality and `noexcept` behavior without requiring its definition. —*end note*]

8.4 Option D

We could fix the two library issues by adding wording to explicitly ignore the accessibility of the destructor, which was the original intent of the first issue, [LWG2116]. Such a change would require support in any compiler intrinsic.

9 Summary

NAD remains the preferred resolution as the current constraint seems be the preferred one; in order to construct an object with dynamic lifetime that can leak without running its destructor, it is necessary to use some form of `new` expression, and that should be a different trait that concerned users can write for themselves if they believe they really need it — noting that `new` operators can be overloaded.

10 Acknowledgements

Thanks to Michael Parks for the pandoc-based framework used to transform this document's source from Markdown.

Thanks again to Matt Godbolt for maintaining Compiler Explorer, the best public resource for C++ compiler and library archaeology!

11 References

[LWG2116] Dave Abrahams. is_nothrow_constructible and destructors.
https://wg21.link/lwg2116

[LWG2827] Richard Smith. is_trivially_constructible and non-trivial destructors.
https://wg21.link/lwg2827

[N1424] John Maddock. A Proposal to add Type Traits to the Standard Library.
https://wg21.link/n1424

[N1836] Matt Austern. 2005-06-24. Draft Technical Report on C++ Library Extensions.
https://wg21.link/n1836

[N3050] D. Abrahams, R. Sharoni, D. Gregor. 2010-03-12. Allowing Move Constructors to Throw (Rev. 1).
https://wg21.link/n3050

[N3142] J. Merrill, D. Krügler, H. Hinnant, G. Dos Reis. 2010-10-15. Adjustments to constructor and assignment traits.
https://wg21.link/n3142

[N4944] Thomas Köppe. 2023-03-22. Working Draft, Standard for Programming Language C++.
https://wg21.link/n4944