Define Delete With Throwing Exception Specification

Addressing one undefined behavior at a time

Document #:	P3424R0
Date:	2024-12-17
Project:	Programming Language C++
Audience:	EWG

Reply-to:	Alisdair Meredith <ameredith1@bloomberg.net>

Contents

1 Abstract

Throwing from an overloaded `delete` operator is undefined behavior, yet `delete` operators have a non-throwing exception specification by default, leading to a deterministic call to `terminate` before any undefined behavior can occur. This paper suggests we can do better than "undefined behavior" for the remaining cases.

2 Revision History

2024 December mailing (post-Wrocław)

— Initial draft of this paper.

3 Introduction

Unless a user provides an explicit `noexcept(false)` exception specification, all deallocation functions have a non-throwing exception specification:

> **14.5 [except.spec] Exception specifications**
>
> 9 A deallocation function (6.7.5.5.3 [basic.stc.dynamic.deallocation]) with no explicit *noexcept-specifier* has a non-throwing exception specification.

This then raises the question of what purpose a potentially throwing exception specification would serve. That question is answered by:

> **6.7.5.5.3 [basic.stc.dynamic.deallocation] Deallocation functions**
>
> 4 If a deallocation function terminates by throwing an exception, the behavior is undefined. The value of …

Whatever else the user intended by allowing an exception to propagate from their deallocation function, the standard is very clear that once you leave that function you proceed straight to undefined behavior, with no window for well-defined behavior where the exception would have turned into a call to `std::terminate` if the default exception specification had been used.

3.1 Basic examples

In preparing this paper, it was observed that the following test code will produce a `false` result for the `noexcept` operator, as-if the implicitly declared deallocation function had a potentially-throwing exception specification:

```
struct T { ~T() noexcept(false); };

T * p = nullptr;
static_assert(noexcept(delete(p)));   // this static_assert fails
```

What is happening is that the destructor for the type `T` is invoked *before* calling the deallocation function, and it is the exception specification on the destructor that is part of the whole expression causing the `noexcept` operator to return `false`. It is impossible to directly test the implicitly declared exception specification for most delete functions in this way.

However, we can extract a `noexcept` test on a deallocation function by testing a *destroying delete* function, which is expected to perform both the destruction of the supplied object and the recovery of any memory associated with that object — typically to support an extended allocation into the region of memory contiguously following said object.

```
#include <new>

struct T {
   ~T() noexcept(false);

   static void operator delete(T*, std::destroying_delete_t);
};

T * p = nullptr;
static_assert(noexcept(delete(p)));
```

3.2 Implementation divergence

Unfortunately, the destroying delete test reveals implementation divergence.

Clang trunk and the EDG compiler follow the Standard specification.

MSVC triggers the `static_assert` because it checks the destructor in this case, even though it should not. If we provide a non-throwing destructor, the test passes, indicating that the implicitly non-throwing exception specification for the destroying delete function is implemented. Also, executing a test program shows that the destructor is never actually called at runtime — this is entirely an artefact of the `noexcept` operator.

Testing against the current trunk for gcc, we see the `static_assert` fires because it does not implement the implicitly non-throwing exception specification for destroying delete. We can then demonstrate the anticipated undefined behavior by throwing an exception from the destroying delete function and catching it — demonstrating that the function is called correctly, and truly lacks the implicit exception specification

4 Proposed resolution

As the only effect of adding a potentially-throwing exception specification to a deallocation function is to allow undefined behavior, we recommend that construct should be disallowed. However, this risks expose the gcc bug of not supplying the implicitly nonthrowing exception to any implementation of a destroying delete function — depending on whether their interpretation would be to make all destroying delete functions without an explicit exception specification ill-formed, or whether the bug would continue to propagate UB. Our hope is that gcc resolve this bug before it becomes an issue, but the timeline is tight if we were to consider this change for C++26.

The question that then remains is whether to allow a redundant `noexcept` or `noexcept(true)` exception specification. This has the potential to break existing code, so we should be cautious about making such syntax immediately ill-formed, although that does seem to be the cleaner long-term solution.

Hence, our recommendation is to make potentially throwing exception specifications outright ill-formed in C++26, and to deprecate non-throwing user-supplied exception specifications on deallocation functions.

5 Wording

All wording is relative to [N5001], the latest working draft at the time of writing. There are no additions to Annex C as the only programs that change behavior and fail to compile previously had undefined behavior.

6.7.5.5.3 [basic.stc.dynamic.deallocation] Deallocation functions

[3] Each deallocation function shall return `void`. If the function is a destroying operator delete declared in class type `C`, the type of its first parameter shall be `C*`; otherwise, the type of its first parameter shall be `void*`. A deallocation function may have more than one parameter. A *usual deallocation function* is a deallocation function whose parameters after the first are

- optionally, a parameter of type `std::destroying_delete_t`, then
- optionally, a parameter of type `std::size_t`,[1] then
- optionally, a parameter of type `std::align_val_t`.

A destroying operator delete shall be a usual deallocation function. A deallocation function may be an instance of a function template. Neither the first parameter nor the return type shall depend on a template parameter. A deallocation function template shall have two or more function parameters. A template instance is never a usual deallocation function, regardless of its signature. A deallocation function shall not have a potentially throwing exception specification (14.5 [except.spec]). A deallocation function with an explicit non-throwing *noexcept-specifier* is deprecated ([depr.except.spec])

[4] ~~If a deallocation function terminates by throwing an exception, the behavior is undefined.~~ The value of the first argument supplied to a deallocation function may be a null pointer value; if so, and if the deallocation function is one supplied in the standard library, the call has no effect.

11.4.11 [class.free] Allocation and deallocation functions

[6] [*Note 3:* If a deallocation function has no explicit *noexcept-specifier*, it has a non-throwing exception specification (14.5 [except.spec]). —*end note*]

14.5 [except.spec] Exception specifications

[9] A deallocation function (6.7.5.5.3 [basic.stc.dynamic.deallocation]) ~~with no explicit *noexcept-specifier*~~ has a non-throwing exception specification.

D.x Deprecated Exception Specification [depr.except.spec]

[1] A deallocation function with an explicit non-throwing *noexcept-specifier* is deprecated (6.7.5.5.3 [basic.stc.dynamic.deallocation]).

[*Note 1*: A deallocation function with a potentially-throwing *noexcept-specifier* is ill-formed —*end note*]

6 Acknowledgements

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

Thanks to Hana Dusíková for calling attention to the interaction with destroying delete.

7 References

[N5001] Thomas Köppe. Working Draft, Programming Languages — C++.
https://wg21.link/n5001

[1] The global `operator delete(void*, std::size_t)` precludes use of an allocation function `void operator new(std::size_t, std::size_t)` as a placement allocation function (C.5.3 [diff.cpp11.basic]).