Please repeat yourself: The noexcept(noexcept(…)) idiom

devblogs.microsoft.com/oldnewthing/20220408-00

April 8, 2022

Raymond Chen

Consider the following class:

```
template<typename T>
struct Holder
{
    T value;

    template<typename... Args>
    Holder(Args&&... args) :
        value(std::forward<Args>(args)...) {}
};

template<typename U> Holder(U&&) ->
    Holder<std::remove_reference_t<U>>
```

The idea is that this class holds a value of type `T`, and it constructs the same way that a `T` does. (The deduction guide lets you write `Holder(42)` instead of `Holder<int>(42)`.)

Now the question: Is the object nothrow constructible?

Nothrow constructibility is important for various operations. For example, a prerequisite for the strong exception guarantee for many operations is that the objects involved be nothrow-copyable or nothrow-movable. And internally, the standard library contains optimizations which are activated if the types involved are nothrow-operable.

Back to the question: Is the object nothrow constructible?

No, this object's constructor is potentially-throwing because it lacks a `noexcept` specifier. This is true even though the construction of the `T` from the `args...` may itself be noexcept.

What we want is for our constructor to be noexcept precisely when the construction of the underlying `T` is noexcept. How can we do that?

The `noexcept` specifier takes an optional compile-time `bool` parameter which indicates whether the function is non-throwing, and the parameter defaults to `true`, which is what you're doing when you write a bare `noexcept`.

Here's the first column of the last table in my discussion of the sad history of the C++ `throw()` exception specifier:

Specifier **C++17**	
Nonthrowing	`noexcept` `noexcept(true)`
	`throw()`
Throwing	`noexcept(false)`
	`throw(something)`

So we can make our constructor conditionally-`noexcept` by putting an appropriate expression inside the parentheses. But what is the correct expression?

The `noexcept` keyword has another purpose: You can use it as an operator in an expression, and it evaluates to `true` if the evaluation of the argument would be considered non-throwing by the compiler. Like `sizeof`, the argument itself is not evaluated.

```
bool example1 = noexcept(1 + 2); // true
bool example2 = noexcept(1 / 0); // true

bool example3 = noexcept(
    std::declval<std::string>().clear()); // true
bool example4 = noexcept(
    std::declval<std::string>().resize(0)); // false
```

The first example is simple: We can add 1 and 2, and there will not be a C++ exception.

The second example is a little trickier: The compiler says that dividing by zero will not raise a C++ exception. Now, dividing by zero is actually undefined behavior, but the compiler isn't performing any division here. It's just checking whether `operator/(int, int)` is potentially-throwing, and it is not.

The third example highlights that the inner expression is not evaluated. We are using the `std::declval<T>` function which pretends to return a `T`, although you are not allowed to actually call it. It may be used only in unevaluated contexts.

The fourth example is a bit interesting: Although resizing a string to zero is functionally equivalent to clearing it, it has a different exception specifier, because the `resize()` method may throw if asked to make a string bigger and it cannot allocate memory for the bigger string.

When the `noexcept(...)` operator is determining whether an expression is potentially-throwing, the compiler looks only at what's printed on the tin.

We can now combine these two lesser-used parts of the `noexcept` keyword. We want our constructor to be conditionally `noexcept` based on whether the inner `value`'s constructor is potentially-throwing given the forwarded `args...`.

Whether the `T` constructor is potentially-throwing given the forwarded `args...` can be calculated by asking the `noexcept(...)` operator to pretend to construct it, and report whether the result is potentially-throwing.

```
noexcept(T(std::forward<Args>(args)...))
```

The result of that calculation is then fed to the `noexcept` specifier to tell it whether the `Holder` constructor, given those arguments, should also be considered potentially-throwing.

```
noexcept(noexcept(T(std::forward<Args>(args)...)))
```

The outer `noexcept` is an exception specifier, but the inner one is a `noexcept` operator. The C++ language is reluctant to introduce new language keywords for fear of breaking existing code that used those potential keywords as identifiers, so it prefers to reuse existing keywords in new ways.[1]

```
template<typename... Args>
Holder(Args&&... args)
    noexcept(noexcept(T(std::forward<Args>(args)...))) :
    value(std::forward<Args>(args)...) {}
```

We want our constructor to have the same potentially-throwing behavior as the construction of `value`, so we use the repetitive `noexcept(noexcept(...))` idiom to say "I'm noexcept if that guy is", and "that guy" is itself a repetition of the thing we're actually going to do one line later.

The `noexcept(noexcept(...))` idiom could be pejoratively called the "Please repeat yourself twice" idiom. You have to repeat the keyword `noexcept`, and you also have to repeat the expression whose potentially-throwing behavior you want to propagate.

Bonus reading: C++ Core Guidelines: The noexcept Specifier and Operator

[1] We saw this in C++11, which took the long-forgotten `auto` keyword and gave it a brand new life.[2]

[2] I'm eagerly awaiting the triumphant return of the `register` keyword, which like `auto` has been reserved since K&R C, but had been largely abandoned, and which was formally stripped of all meaning in C++17.

Raymond Chen

Follow