The sad history of the C++ throw(…) exception specifier

devblogs.microsoft.com/oldnewthing/20180928-00

September 28, 2018

Raymond Chen

When exceptions were introduced into the C++ language, a corresponding `throw(...)` dynamic exception specifier was introduced which annotated which exceptions could be throw by a function.

```
// this function might throw an integer or a pointer to char,
// but nothing else.
void foo() throw(int, char*);
```

This has made a lot of people very angry and has been widely regarded as a bad move.

According to the C++98 standard, if a function throws an exception not listed among the types specified in its dynamic exception specifier, the system called the `std::unexpected()` function, and the default behavior of `std::unexpected()` is to terminate the program by calling `std::terminate()`. As a special case, `throw()` means that the function shouldn't throw any exceptions at all.

By C++11, the `throw(...)` dynamic exception specifier was deprecated, and in C++17, all support for dynamic exception specifiers was removed save for the special case of `throw()`. At the same time, they changed the penalty for throwing an exception when you said you wouldn't: the runtime calls `std::terminate()` directly, rather than passing through `std::unexpected()`.

But of course the Microsoft C++ compiler has to do things a little bit differently.

Specifier C++14 and earlier		**Disallowed exception thrown**	
		Standard behavior	**Microsoft behavior**
Nonthrowing	`noexcept` `noexcept(true)`	`std::` `unexpected`	`std::` `unexpected`
	`throw()`	`std::` `unexpected`	undefined behavior ⇐
Throwing	`noexcept(false)`	exception propagates	exception propagates

	`throw(something)`	`std::unexpected`	exception propagates ⇐

The Microsoft C++ compiler treats the `throw(...)` exception specifier as a promise on the part of the programmer, but there is no enforcement. It trusts you to adhere to your self-imposed contract. If an exception is thrown when the function promised that no exceptions would be thrown, the behavior is undefined. If the function said that some exceptions could be thrown, the compiler doesn't validate that the actual thrown exception is allowed; it just propagates the exception.

In practice, what happened is that the compiler performed optimizations on the assumption that no disallowed exception would be thrown. The most common such optimization is that the compiler won't bother registering unwind codes for things that it "knows" will never require unwinding because there are no points where an exception could be thrown prior to the object's destruction.

```
void Example()
{
   ObjectWithDestructor obj;
   obj.stuff_that_does_not_throw();
   // destructor runs here
}
```

If `stuff_that_does_not_throw` is marked as non-throwing, then the compiler can avoid having to register `obj` for unwinding during exception propagation, since you promised that no exception could escape.

And then you throw an exception and invalidate all those optimizations. The most common visible effect of this is that an exception propagated out of a function that should never have let an exception escape, and some object destructors failed to run.

But wait, all is not lost.

If you enable `/std:c++17`, then the Microsoft C++ compiler will implement the standard behavior for `throw(...)`.

Specifier C++17		**Disallowed exception thrown**	
		Standard behavior	**Microsoft behavior with `/std:c++17`**
Nonthrowing	`noexcept` `noexcept(true)`	`std::terminate`	`std::terminate`
	`throw()`	`std::terminate`	`std::terminate`
Throwing	`noexcept(false)`	exception propagates	exception propagates

	`throw(something)`	not supported	not supported

Yes, it took a long time to get there, but better late than never.

Raymond Chen

Follow