20.9 — Exception specifications and noexcept

ALEX SEPTEMBER 21, 2021

(h/t to reader Koe for providing the first draft of this lesson!)

In C++, all functions are classified as either **non-throwing** (do not throw exceptions) or **potentially throwing** (may throw an exception).

Consider the following function declaration:

```
1 int doSomething(); // can this function throw an exception or
  not?
```

Looking at a typical function declaration, it is not possible to determine whether a function might throw an exception or not. While comments may help enumerate whether a function throws exceptions or not (and if so, what kind of exceptions), documentation can grow stale and there is no compiler enforcement for comments.

Exception specifications are a language mechanism that was originally designed to document what kind of exceptions a function might throw as part of a function specification. While most of the `exception specifications` have now been deprecated or removed, one useful `exception specification` was added as a replacement, which we'll cover in this lesson.

The noexcept specifier

The **noexcept specifier** defines a function as `non-throwing`. To define a function as `non-throwing`, we can use the `noexcept` specifier in the function declaration, placed to the right of the function parameter list:

```
1 void doSomething() noexcept; // this function is non-
  throwing
```

Note that `noexcept` doesn't actually prevent the function from throwing exceptions or calling other functions that are `potentially throwing`. Rather, when an exception is thrown, if an exception exits a `noexcept` function, `std::terminate` will be called. And note that if `std::terminate` is called from inside a `noexcept` function, stack unwinding may or may not occur (depending on implementation and optimizations), which means your objects may or may not be destructed properly prior to termination.

Much like functions that differ only in their return values can not be overloaded, functions differing only in their exception specification can not be overloaded.

The noexcept specifier with a Boolean parameter

The `noexcept` specifier has an optional Boolean parameter. `noexcept(true)` is equivalent to `noexcept`, meaning the function is `non-throwing`. `noexcept(false)` means the function is potentially throwing. These parameters are typically only used in template functions, so that a template function can be dynamically created as `non-throwing` or `potentially throwing` based on some parameterized value.

Which functions are non-throwing and potentially-throwing

Functions that are `non-throwing` by default:

- default constructors
- copy constructors
- move constructors
- destructors
- copy assignment operators
- move assignment operators

However, if any of the listed functions call (explicitly or implicitly) another function which is `potentially throwing`, then the listed function will be treated as `potentially throwing` as well. For example, if a class has a data member with a `potentially throwing` constructor, then the class's constructors will be treated as `potentially throwing` as well. As another example, if a copy assignment operator calls a `potentially throwing` assignment operator, then the copy assignment will be `potentially throwing` as well.

Best practice

If you want any of the above listed functions to be non-throwing, explicitly tag them as `noexcept` (even though they are defaulted that way), to ensure they don't inadvertently become `potentially throwing`.

The following are `potentially throwing` by default:

- Normal functions
- User-defined constructors
- Some operators, such as `new`

The noexcept operator

The `noexcept` operator can be used inside functions. It takes an expression as an argument, and returns `true` or `false` if the compiler thinks it will throw an exception or not. The `noexcept` operator is checked statically at compile-time, and doesn't actually evaluate the input expression.

```
1  void foo() {throw -1;}
   void boo() {};
2  void goo() noexcept {};
3  struct S{};

4  constexpr bool b1{ noexcept(5 + 3) }; // true; ints are non-throwing
5  constexpr bool b2{ noexcept(foo()) }; // false; foo() throws an exception
6  constexpr bool b3{ noexcept(boo()) }; // false; boo() is implicitly noexcept(false)
   constexpr bool b4{ noexcept(goo()) }; // true; goo() is explicitly noexcept(true)
   constexpr bool b5{ noexcept(S{}) };   // true; a struct's default constructor is noexcept by
   default
```

The `noexcept` operator can be used to conditionally execute code depending on whether it is `potentially throwing` or not. This is

required to fulfill certain **exception safety guarantees**, which we'll talk about in the next section.

Exception safety guarantees

An **exception safety guarantee** is a contractual guideline about how functions or classes will behave in the event an exception occurs. There are four levels of exception safety:

- No guarantee -- There are no guarantees about what will happen if an exception is thrown (e.g. a class may be left in an unusable state)
- Basic guarantee -- If an exception is thrown, no memory will be leaked and the object is still usable, but the program may be left in a modified state.
- Strong guarantee -- If an exception is thrown, no memory will be leaked and the program state will not be changed. This means the function must either completely succeed or have no side effects if it fails. This is easy if the failure happens before anything is modified in the first place, but can also be achieved by rolling back any changes so the program is returned to the pre-failure state.
- No throw / No fail -- The function will always succeed (no-fail) or fail without throwing an exception (no-throw).

Let's look at the no-throw/no-fail guarantees is more detail:

The `no-throw guarantee`: if a function fails, then it won't throw an exception. Instead, it will return an error code or ignore the problem. No-throw guarantees are required during stack unwinding when an exception is already being handled; for example, all destructors should have a no-throw guarantee (as should any functions those destructors call). Examples of code that should be no-throw:

- destructors and memory deallocation/cleanup functions
- functions that higher-level no-throw functions need to call

The `no-fail guarantee`: a function will always succeed in what it tries to do (and thus never has a need to throw an exception, thus, no-fail is a slightly stronger form of no-throw). Examples of code that should be no-fail:

- move constructors and move assignment (move semantics, covered in chapter M)
- swap functions
- clear/erase/reset functions on containers
- operations on std::unique_ptr (also covered in chapter M)
- functions that higher-level no-fail functions need to call

When to use noexcept

Just because your code doesn't explicitly throw any exceptions doesn't mean you should start sprinkling `noexcept` around your code. By default, most functions are `potentially throwing`, so if your function calls other functions, there is a good chance it calls a function that is `potentially throwing`, and thus is `potentially throwing` too.

The standard library's policy is to use `noexcept` only on functions that *must not* throw or fail. Functions that are `potentially throwing` but do not actually throw exceptions (due to implementation) typically are not marked as `noexcept`.

Best practice

Use the `noexcept` specifier in specific cases where you want to express a no-fail or no-throw guarantee.

Best practice

If you are uncertain whether a function should have a no-fail/no-throw guarantee, error on the side of caution and do not mark it with `noexcept`. Reversing a decision to use `noexcept` violates an interface commitment to the user about the behavior of the function. Making guarantees stronger by retroactively adding `noexcept` is considered safe.

Why it's useful to mark functions as non-throwing

There are a few good reasons to mark functions a non-throwing:

- Non-throwing functions can be safely called from functions that are not exception-safe, such as destructors
- Functions that are **noexcept** can enable the compiler to perform some optimizations that would not otherwise be available. Because a `noexcept` function cannot throw an exception outside the function, the compiler doesn't have to worry about keeping the runtime stack in an unwindable state, which can allow it to produce faster code.
- There are also a few cases where knowing a function is `noexcept` allows us to produce more efficient implementations in our own code: the standard library containers (such as std::vector) are `noexcept` aware and will use the `noexcept` operator to determine whether to use `move semantics` (faster) or `copy semantics` (slower) in some places (we cover `move semantics` in chapter M).

Dynamic exception specifications

Optional reading

Before C++11, and until C++17, `dynamic exception specifications` were used in place of `noexcept`. The **dynamic exception specifications** syntax uses the `throw` keyword to list which exception types a function might directly or indirectly throw:

```
1 int doSomething() throw(); // does not throw exceptions
  int doSomething() throw(std::out_of_range, int*); // may throw either std::out_of_range or a pointer
  to an integer
2 int doSomething() throw(...); // may throw anything
```

Due to factors such as incomplete compiler implementations, some incompatibility with template functions, common misunderstandings about how they worked, and the fact that the standard library mostly didn't use them, the dynamic exception specifications were deprecated in C++11 and removed from the language in C++17 and C++20. See this paper for more context.

Next lesson

20.x Chapter 20 comprehensive quiz

Back to table of contents

Previous lesson

20.8 Exception dangers and downsides

Leave a comment... Put C++ code between triple-backticks (markdown style):```Your C++ coc

Name*

Email*

Avatars from https://gravatar.com/ are connected to your provided email address.

Notify me about replies: POST COMMENT

DP N N FOUT

Newest