27.9 — Exception specifications and noexcept

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

Looking at a typical function declaration, it is not possible to determine whether a function might throw an exception or not:

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

In the above example, can the `doSomething` function throw an exception? It's not clear. But the answer is important in some contexts. If `doSomething` could throw an exception, then it's not safe to call this function from a destructor, or any other place where a thrown exception is undesirable.

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

In C++, all functions are classified as either *non-throwing* or *potentially throwing*. A **non-throwing function** is one that promises not to throw exceptions that are visible to the caller. A **potentially throwing function** may throw exceptions that are visible to the caller.

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

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

Note that `noexcept` doesn't actually prevent the function from throwing exceptions or calling other functions that are potentially throwing. This is allowed so long as the noexcept function catches and handles those exceptions internally, and those exceptions do not exit the noexcept function.

If an unhandled exception would exit a noexcept function, `std::terminate` will be called (even if there is an exception handler that would otherwise handle such an exception somewhere up the stack). And 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.

Key insight

The promise that a noexcept function makes to not throw exceptions that are visible to the caller is a contractual promise, not a promise enforced by the compiler. So while calling a noexcept function should be safe, any exception handling bugs in the noexcept function that cause the contract to be broken will result in termination of the program! This shouldn't happen, but neither should bugs.

For this reason, it's best that noexcept functions don't mess with exceptions at all, or call potentially throwing functions that could raise an exception. A noexcept function can't have an exception handling bug if no exceptions can possibly be raised in the first place!

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.

Illustrating the behavior of noexcept functions and exceptions

The following program illustrates the behavior of noexcept functions and exceptions in various cases:

```
// h/t to reader yellowEmu for the first draft of this program
#include <iostream>

class Doomed
{
public:
    ~Doomed()
    {
        std::cout << "Doomed destructed\n";
    }
};

void thrower()
{
    std::cout << "Throwing exception\n";
    throw 1;
}

void pt()
{
    std::cout << "pt (potentally throwing) called\n";
    //This object will be destroyed during stack unwinding (if it occurs)
    Doomed doomed{};
    thrower();
    std::cout << "This never prints\n";
}

void nt() noexcept
{
    std::cout << "nt (noexcept) called\n";
    //This object will be destroyed during stack unwinding (if it occurs)
    Doomed doomed{};
    thrower();
    std::cout << "this never prints\n";
}

void tester(int c) noexcept
{
    std::cout << "tester (noexcept) case " << c << " called\n";
    try
    {
        (c == 1) ? pt() : nt();
    }
    catch (...)
    {
        std::cout << "tester caught exception\n";
    }
}

int main()
{
    std::cout << std::unitbuf; // flush buffer after each insertion
```

```
    std::cout << std::boolalpha; // print boolean as true/false
    tester(1);
    std::cout << "Test successful\n\n";
    tester(2);
    std::cout << "Test successful\n";

    return 0;
}
```

On the author's machine, this program printed:

```
tester (noexcept) case 1 called
pt (potentially throwing) called
Throwing exception
Doomed destructed
tester caught exception
Test successful

tester (noexcept) case 2 called
nt (noexcept) called
throwing exception
terminate called after throwing an instance of 'int'
```

and then the program aborted.

Let's explore what's happening here in more detail. Note that `tester` is a noexcept function, and thus promises not to expose any exception to the caller (`main`).

The first case illustrates that noexcept functions can call potentially throwing functions and even handle any exceptions those functions throw. First, `tester(1)` is called, which calls potentially throwing function `pt` , which calls `thrower`, which throws an exception. The first handler for this exception is in `tester`, so the exception unwinds the stack (destroying local variable `doomed` in the process), and the exception is caught and handled within `tester`. Because `tester` does not expose this exception to the caller (`main`), there is no violation of noexcept here, and control returns to `main`.

The second case illustrates what happens when a noexcept function tries to pass an exception back to its caller. First, `tester(2)` is called, which calls non-throwing function `nt`, which calls `thrower`, which throws an exception. The first handler for this exception is in `tester`. However, `nt` is noexcept, and to get to the handler in `tester`, the exception would have to propagate to the caller of `nt`. That is a violation of the noexcept of `nt`, and so `std::terminate` is called, and our program is aborted immediately. On the author's machine, the stack was not unwound (as illustrated by `doomed` not being destroyed).

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 implicitly non-throwing:

Destructors

Functions that are non-throwing by default for implicitly-declared or defaulted functions:

- Constructors: default, copy, move
- Assignments: copy, move
- Comparison operators (as of C++20)

However, if any of these 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.

Functions that are potentially throwing (if not implicitly-declared or defaulted):

- Normal functions
- User-defined constructors
- User-defined operators

The noexcept operator

The noexcept operator can also be used inside expressions. 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.

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

constexpr bool b1{ noexcept(5 + 3) }; // true; ints are non-throwing
constexpr bool b2{ noexcept(foo()) }; // false; foo() throws an exception
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 guarantees:

- 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 guarantee -- 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 in 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 22)
- swap functions
- clear/erase/reset functions on containers
- operations on std::unique_ptr (also covered in chapter 22)
- 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.

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 significant 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 22, and this optimization in lesson 27.10 -- std::move_if_noexcept.

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`.

For your own code, always mark the following as `noexcept`:

- Move constructors
- Move assignment operators
- Swap functions

For your code, consider marking the following as `noexcept`:

- Functions for which you want to express a no-throw or no-fail guarantee (e.g. to document that they can be safely called from destructors or other noexcept functions)
- Copy constructors and copy assignment operators that are no-throw (to take advantage of optimizations).
- Destructors. Destructors are implicitly noexcept so long as all members have noexcept destructors

Best practice

Always make move constructors, move assignment, and swap functions `noexcept`.

Make copy constructors and copy assignment operators `noexcept` when you can.

Use `noexcept` on other functions 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, err 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, and may break existing code. Making guarantees stronger by later adding noexcept to a function that was not originally noexcept is considered safe.

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:

```
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
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.