Item 19: Declare functions `noexcept` whenever possible.

In C++98, exception specifications were rather temperamental beasts. You had to summarize the exception types a function might emit, and this imposed constraints on the function's implementation. If the implementation was changed, the exception specification might have to be updated, too, and that not only opened the door to consistency errors (i.e., an exception specification that no longer corresponded to the implementation), it also meant that callers might stop compiling, because an exception specification is part of a function's interface. For these reasons, C++98 exception specifications never gained much popularity. Developers and libraries generally shied away from them, and some compilers didn't even fully implement them.

Over time, a consensus emerged that the only meaningful information about a function's exception-emitting behavior was whether it had any. Black or white: either a function might emit an exception or it guaranteed that callers would never see one. This maybe-or-never dichotomy forms the basis of C++11's exception specifications, which essentially replace C++98's. (C++98-style exception specifications continue to be legal in C++11 and C++14, but they're deprecated.) In C++11, `noexcept` is for functions that guarantee they'll never emit an exception. When you write a function that can make that guarantee, you'll want to use `noexcept`. Why? Because it permits compilers to generate better code.

There are two reasons for this, but we'll begin with how C++98's way of saying "this function emits no exceptions" differs from C++11's.

Suppose we have a function `f` that promises callers they'll never receive an exception. The C++98 and C++11 ways of expressing that are:

```
int f(int x) throw();       // C++98 approach: f emits no
                            // exceptions

int f(int x) noexcept;      // C++11 approach: f emits no
                            // exceptions
```

Perhaps surprisingly, neither C++98 nor C++11 permits compilers to reject code in `f` that could violate these exception specifications. As a result, `f` could be implemented like this:

```
int f(int x) noexcept      // C++98 version would use "throw()"
{
  if (x >= 0) return x * x - 42;                  // if x >= 0 …

  throw std::invalid_argument(                    // else throw!
    "Invalid value for x: " + std::to_string(x)
  );
}
```

This may look absurd, but it’s perfectly legal C++. Furthermore, looks aren’t everything. The code here could be a way of enforcing the precondition that `x` must be non-negative. If `f` is called with a legitimate value, it doesn't throw. However, if an invalid value is passed in, the precondition violation causes the function to have undefined behavior, and this implementation uses that freedom—undefined behavior means that *anything* can happen—to throw an exception.

As an aside, note the use of `std::to_string` to produce a textual representation of the value of `x`. Among C++11’s lesser-known features is a set of overloaded `std::to_string` functions that produce `std::string` objects from numeric values. The Standard Library has functions to perform the reverse transformations, too (i.e., from `std::strings` to `ints`, `unsigneds`, `doubles`, etc.), but the naming convention for those functions, albeit following a consistent pattern, is rather cryptic: `stoi`, `stol`, `stod`, etc. The C++11 Standard Library also offers `std::wstring`-based versions of all these functions.

But back to the difference in meaning between these two declarations:

```
int f(int x) throw();              // C++98 approach

int f(int x) noexcept;             // C++11 approach
```

If, at runtime, an exception leaves `f`, `f`’s exception specification is violated. With the C++98 approach, the call stack is unwound to `f`’s caller, then the *unexpected handler* function is invoked, and that leads to program termination (typically by calling `std::terminate`). With the C++11 approach, runtime behavior is slightly different: the stack is *possibly* unwound, then program execution is terminated (typically by calling `std::terminate`).

The fact that, with `noexcept`, the call stack only *might* be unwound turns out to make a big difference during code generation. Optimizers are no longer constrained to keep the runtime stack in an unwindable state if an exception would propagate out of the function, nor must they ensure that objects in a `noexcept` function are destroyed in the inverse order of construction should an exception leave the function. The result is greater opportunities for optimization, not only within the body of a `noexcept` function, but also at call sites to the function. This degree of flexibility is present only for `noexcept` functions. Functions with "`throw()`" exception specifications lack it, as do functions with no exception specification at all. The situation can be summarized this way (where it doesn't make any difference what `func` does):

```
RetType func(parameters) noexcept;      // more optimizable
RetType func(parameters) throw();       // less optimizable
RetType func(parameters);               // less optimizable
```

This alone should provide sufficient motivation to declare functions `noexcept` whenever you can. For some functions, however, the case is even stronger.

The move operations are the preeminent example. Suppose you have a large investment in a C++98 code base making use of `std::vector`s of `Widget`s. `Widget`s are added to the `std::vector`s from time to time, perhaps via `push_back`:

```
std::vector<Widget> vw;
…
Widget w;
…                          // put w into proper state
                           // for addition to vw
vw.push_back(w);           // add w to vw
…
```

Assume this code works fine, and you have no interest in modifying it for C++11. However, you do want to take advantage of the fact that C++11's move semantics can improve the performance of existing code when move-enabled types are involved. You therefore ensure that `Widget` has move operations, either by writing them yourself or by seeing to it that the conditions for their automatic generation are fulfilled (see Item 20).

When a new element is added to a `std::vector` via `push_back`, it's possible that the `std::vector` lacks space for it, i.e., that the `std::vector`'s size is equal to its capacity. When that happens, the `std::vector` allocates a new, larger, chuck of memory to hold its elements, and it transfers the elements from the existing chunk of memory to the new one. In C++98, the transfer was accomplished by copying each element from the old memory into the new memory, then destroying the copies in the old memory. This approach enabled `push_back` to offer the strong exception safety guarantee: if an exception was thrown during the copying of the elements, the state of the `std::vector` remained unchanged, because none of the elements in the original memory was destroyed until all elements had been successfully copied into the new memory.

In C++11, a natural optimization would be to replace the copying of `std::vector` elements with moves. Unfortunately, doing this runs the risk of violating `push_back`'s exception safety guarantee. If *n* elements have been moved from old memory to new and an exception is thrown moving element *n+1*, the `push_back` operation can't run to completion. But the original `std::vector` has been modified: *n* of its elements have been moved from. Restoring their original state may not be possible, because attempting to move each object back into the original memory may itself yield an exception.

This is a serious problem, because the behavior of your C++98 code base could depend on `push_back`'s strong exception safety guarantee. C++11 implementations therefore can't silently replace copy operations inside `push_back` with moves. They must continue to employ copy operations. *Unless*, that is, it's known that the move operations are guaranteed not to emit exceptions. In that case, replacing element copy operations inside `push_back` with move operations would be safe, and the only side effect would be improved performance.

std::vector::push_back takes advantage of this "move if you can, but copy if you must" strategy, and it's not the only function in the Standard Library that does. Other functions sporting the strong exception safety guarantee in C++98 (e.g., std::vector::reserve, std::deque::insert, etc.) behave the same way. All these functions replace calls to copy operations in C++98 with calls to move operations in C++11 if (and only if) the move operations are known to never emit exceptions. But how can a compiler know if a move operation won't produce an exception? The answer is obvious: it checks to see if the operation is declared noexcept.*

And yet, it's not quite that simple. Popping the hood and peeking inside to see how things work is instructive, so here we go.

Inside a function like push_back, suppose we want to transfer an object from one location in memory to another (i.e., copy or move it, depending on what is appropriate). Assume we have an iterator, src, referring to the object to be transferred and a second iterator, dest, referring to where it should go. Our code would have a statement like this:

```
*dest = *src;                         // transfer *src to *dest;
                                      // this is incorrect
```

The statement would be inside a loop, because we'd ultimately need to transfer all the objects in the container, but understanding how things work for one object is all we need for this discussion.

As the comment indicates, the code is incorrect. The problem is that *src is an lvalue, so this statement would unconditionally copy *src to *dest. That'd have been fine in C++98, but in C++11, we want to do a move if we can. The usual way to move an lvalue is to apply std::move to it:

* Alternatively, the function could have a C++98-style empty exception specification (i.e., "throw()"), but the only reason I can imagine why a move operation—something that didn't exist in C++98 and therefore can't be part of a legacy code base—would employ throw() instead of noexcept would be to accommodate compilers with incomplete C++11 support, i.e., compilers where move operations are supported, but noexcept isn't.

```
*dest = std::move(*src);                // transfer *src to *dest;
                                        // this is incorrect
```

This is also incorrect, because now we're moving `*src`, regardless of whether its move assignment operator is `noexcept`. As we've discussed, doing that would prevent `push_back` from maintaining its strong exception safety guarantee, and maintaining that guarantee is essential for ensuring that code written under C++98 continues to function properly.

The correct code takes advantage of `std::move`'s poorly publicized cousin, `std::move_if_noexcept`:

```
*dest = std::move_if_noexcept(*src);  // transfer *src to *dest;
                                      // this is correct
```

Conceptually, `std::move_if_noexcept` causes `*src` to be moved if its move assignment operator is `noexcept`, and otherwise it causes `*src` to be copied. That's exactly what we want, and that's why this code is correct. Uses of `std::move_if_noexcept` are scattered throughout strongly exception safe functions in the Standard Library, and that's why you have a special incentive to declare your move operations `noexcept`: it enables their use inside such functions.

The conceptual description of `std::move_if_noexcept` deviates from its true behavior in two small ways. First, if `std::move_if_noexcept` is invoked on an object of a move-only type, a move will be performed, even if it might yield an exception. This is understandable: what else can `std::move_if_noexcept` do, given that the type can't be copied? Besides, this behavior can't break legacy code, because there was no such thing as a move-only type in C++98.

Second, `std::move_if_noexcept`, like `std::move`, doesn't actually move anything. Rather, it performs a cast to an rvalue that, through overloading resolution, can cause a move assignment operator or a move constructor to be invoked. It's these functions that actually move values around. For details on the relationship among `std::move` (and `std::move_if_noexcept`), casting to rvalues, move operations, and overload resolution, consult Item 21.

Accompanying the move operations on the list of functions where a `noexcept` declaration is especially beneficial is `swap`. Being a heavily-used function, the optimization opportunities that `noexcept` affords are unusually worthwhile. (Many algorithms rely on `swap`, as do implementations of many copy assignment operators.) Furthermore, whether particular versions of `swap` in the Standard Library are `noexcept` is sometimes dependent on whether user-defined type-specific `swap`s are `noexcept`. For example, the declarations for the Standard Library's `swap`s for arrays and for `std::pair` are:

```
template <class T, size_t N>
void swap(T (&a)[N],
          T (&b)[N]) noexcept(noexcept(swap(*a, *b)));

template <class T1, class T2>
struct pair {
  …
  void swap(pair& p) noexcept(noexcept(swap(first, p.first)) &&
                              noexcept(swap(second, p.second)));
  …
};
```

These functions are *conditionally noexcept*: whether they are `noexcept` depends on whether the expressions inside the `noexcept`s are. Given two arrays of `Widget`, for example, `swap`ping them is `noexcept` only if `swap`ping individual elements from the arrays is `noexcept`, i.e., if `swap` for `Widget` is `noexcept`. The author of `Widget`'s `swap` thus determines whether `swap`ping arrays of `Widget` is `noexcept` (which, in turn, could determine whether other `swap`s are `noexcept`, e.g., `swap` for arrays of arrays of `Widget`). Similarly, whether `swap`ping two `std::pair` objects containing `Widget`s is `noexcept` depends on whether `swap` for `Widget`s is `noexcept`.

The fact that swapping higher-level data structures can generally be `noexcept` only if swapping their lower-level constituents is `noexcept` is the reason why you should strive to offer `noexcept` `swap` functions.

Because `noexcept` is part of a function's interface, you should declare a function `noexcept` only if you are willing to commit to a `noexcept` implementation over the long term. If you declare a function `noexcept` and implement it accordingly, then later decide you wish you hadn't made the `noexcept` promise, your options are bleak. You can remove `noexcept` from the function's declaration, thus running the risk of breaking arbitrary amounts of client code. You can retain the `noexcept`

Comment [sdm1]: Font should be both code and Term Introduction.

declaration, but change the implementation such that an exception could actually escape. In that case, if an exception did escape at runtime, your program would be terminated. Or you can retain your existing implementation, thus abandoning whatever motivated you to want to change the implementation in the first place. None of these options is appealing.

Most functions are *exception-neutral*: they don't throw exceptions themselves, but if a function they call produces one (directly or indirectly), the exception causes no harm as it passes through on its way to a hander in a different function. Exception-neutral functions aren't `noexcept`, because exceptions may pass through them. Most functions, therefore, aren't `noexcept`.

Some functions, however, are naturally `noexcept`, and for a few more—notably the move operations and `swap`—being `noexcept` has such a significant payoff, it's worth implementing them in a `noexcept` manner if at all possible. When you can honestly say that a function should never emit exceptions, you should definitely declare it `noexcept`.

Things to Remember

- `noexcept` functions offer more optimization opportunities than non-`noexcept` functions.
- C++98 functions offering the strong exception safety guarantee may internally call `std::move_if_noexcept` instead of `std::move`.
- Strive to declare the move operations and `swap` `noexcept`.