Item 16: Declare functions noexcept whenever possible.

In C++98, exception specifications were rather temperamental creatures. You had to summarize the exception types a function might emit, so if the function's implementation was modified, the exception specification might need revision, too. Changing an exception specification could break client code, because callers might be dependent on the original exception specification. Compilers typically offered no help in maintaining consistency among function implementations, exception specifications, and client code. Most programmers ultimately decided that C++98 exception specifications weren't worth the trouble.

Interest in the idea of exception specifications remained strong, however, and as work on C++ progressed, a consensus emerged that the truly 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 it wouldn't. 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 remain valid, but they're deprecated.) In C++11, `noexcept` is for functions that guarantee they won't emit an exception.

Whether a function should be so declared is fundamentally a matter of interface design. The exception-emitting behavior of a function is of key interest to clients. Callers can query a function's `noexcept` status, and the results of such a query can affect the exception safety or efficiency of the calling code. As such, whether a function is `noexcept` is as important a piece of information as whether a member function is `const`. Failure to declare a function `noexcept` when you know that it will never emit an exception is simply poor interface specification.

But there's an additional incentive to apply `noexcept` to functions that won't produce exceptions: it permits compilers to generate better object code. To understand why, it helps to examine the difference between the C++98 and C++11 ways of saying that a function won't emit exceptions. Consider a function `f` that promises callers they'll never receive an exception. The two ways of expressing that are:

```
int f(int x) throw();     // no exceptions from f: C++98 style
```

```
int f(int x) noexcept;    // no exceptions from f: C++11 style
```

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, and, after some actions not relevant here, program execution is terminated. With the C++11 approach, runtime behavior is a bit different: the stack is only *possibly* unwound before program execution is terminated.

The difference between unwinding the call stack and *possibly* unwinding it has a surprisingly large impact on code generation. In a `noexcept` function, optimizers need not 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 more opportunities for optimization, not only within the body of a `noexcept` function, but also at sites where the function is called. Such 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:

```
RetType function(params) noexcept;     // most optimizable
RetType function(params) throw();      // less optimizable
RetType function(params);              // less optimizable
```

This alone should provide sufficient motivation to declare functions `noexcept` whenever you can.

For some functions, the case is even stronger. The move operations are the preeminent example. Suppose you have 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;
…                          // work with w
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 legacy 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 19).

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, chunk 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 to the new memory, then destroying the originals 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 the old memory 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 legacy code could depend on `push_back`'s strong exception safety guarantee. Therefore, C++11 implementations 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 not emit exceptions. But how can a function know if a move operation won't produce an exception? The answer is obvious: it checks to see if the operation is declared `noexcept`.*

`swap` functions comprise another case where `noexcept` is particularly desirable. `swap` is a key component of many STL algorithm implementations, and it's commonly employed in copy assignment operators, too. Its widespread use renders the optimizations that `noexcept` affords especially worthwhile. Furthermore, whether `swap`s in the Standard Library are `noexcept` is sometimes dependent on whether user-defined `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)));
  …
};
```

* The checking is typically rather roundabout. Functions like `std::vector::push_back` call `std::move_if_noexcept`, a variation of `std::move` that conditionally casts to an rvalue (see Item 28), depending on whether the type's move constructor is `noexcept`. In turn, `std::move_if_noexcept` calls `std::is_nothrow_move_constructible`, and the value of this type trait is set by compilers, based on whether the move constructor has a `noexcept` (or `throw()`) designation.

These functions are *conditionally noexcept*: whether they are `noexcept` depends on whether the expressions inside the `noexcept`s are `noexcept`. 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`. That, in turn, determines whether other `swap`s, such as the one for arrays of arrays of `Widget`, are `noexcept`. 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.

Comment [sdm1]: Style: Code + Term Introduction.

By now, I hope you're excited about the optimization opportunities that `noexcept` affords. Alas, I must temper your enthusiasm. Optimization is important, but correctness is more important. I noted at the beginning of this Item that `noexcept` is part of a function's interface, so 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 later regret that decision, your options are bleak. You can remove `noexcept` from the function's declaration (i.e., change its interface), thus running the risk of breaking client code. You can change the implementation such that an exception could escape, but keep the original (now incorrect) exception specification. If you do that, your program will be terminated if an exception tries to leave the function. Or you can resign yourself to your existing implementation, abandoning whatever motivated your desire to change the implementation in the first place. None of these options is appealing.

The fact of the matter is that most functions are *exception-neutral*. Such functions throw no exceptions themselves, but functions they call might emit one. When that happens, the calling function allows the emitted exception to pass through on its way to a handler further up the call chain. Exception-neutral functions are never `noexcept`, because they may emit such "just passing through" exceptions. Most functions, therefore, quite properly lack the `noexcept` designation.

Some functions, however, have natural implementations that emit no exceptions, 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`.

Comment [sdm2]: Should I say that constexpr functions are typically good candidates for noexcept?

Please note that I said some functions have *natural* `noexcept` implementations. Twisting a function's implementation to permit a `noexcept` declaration is the tail wagging the dog. Is putting the cart before the horse. Is not seeing the forest for the trees. Is…choose your favorite metaphor. If a straightforward function implementation might yield exceptions (e.g., by invoking a function that might throw), the hoops you'll jump through to hide that from callers (e.g., catching all exceptions and replacing them with status codes or special return values) will not only complicate your function's implementation, it will typically complicate code at call sites, too (e.g., code there may have to check for status codes or special return values). The runtime cost of those complications (e.g., extra branches, larger functions that put more pressure on instruction caches, etc.) could exceed any speedup you'd hope to achieve via `noexcept`, plus you'd be saddled with source code that's more difficult to comprehend and maintain. That'd hardly be exemplary software engineering. As a general rule, the only time it makes sense to actively search for a `noexcept` algorithm is when you're implementing the move functions or `swap`.

Two more points about `noexcept` functions are worth mentioning. First, in C++98, it was considered bad style to permit the memory deallocation functions (i.e., `operator delete` and `operator delete[]`) and destructors to emit exceptions, and in C++11, this style rule has been all but upgraded to a language rule. By default, all memory deallocation functions and all destructors—both user-defined and com-

† The prescribed declarations for move operations on containers in the Standard Library lack `noexcept`. However, implementers are permitted to strengthen exception specifications for Standard Library functions, and, in practice, it is common for at least some container move operations to be declared `noexcept`. That practice exemplifies this Item's advice. Having found that it's possible to write container move operations such that exceptions never need to be emitted, implementers often declare the operations `noexcept`, even though the Standard does not require them to do so.

piler-generated—are implicitly `noexcept`. There's thus no need to declare them `noexcept`. (Doing so doesn't hurt anything, it's just unconventional.) The only time a destructor is not implicitly `noexcept` is when a data member of the class (including inherited members and those contained inside other data members) is of a type that expressly states that its destructor may emit exceptions (e.g., declares it "`noexcept(false)`"). Such destructors are uncommon. There are none in the Standard Library.

Second, let me elaborate on my earlier observation that compilers typically offer no help in identifying inconsistencies between function implementations and their exception specifications. Consider this code, which is perfectly legal:

```
void setup();            // functions defined elsewhere
void cleanup();

void doWork() noexcept
{
  setup();               // set up work to be done

  …                      // do the actual work

  cleanup();             // perform cleanup actions
}
```

Here, `doWork` is declared `noexcept`, even though it calls the non-`noexcept` functions `setup` and `cleanup`. This seems contradictory, but it could be that `setup` and `cleanup` document that they never emit exceptions, even though they're not declared that way. There could be good reasons for their non-`noexcept` declarations. For example, they might be part of a library written in C. (Even functions from the C Standard Library that have been moved into the `std` namespace lack exception specifications, e.g., `std::strlen` isn't declared `noexcept`.) Or they could be part of a C++98 library that decided not to use C++98 exception specifications and hasn't yet been revised for C++11.

Because there are legitimate reasons for `noexcept` functions to rely on code lacking the `noexcept` guarantee, C++ permits such code, and compilers generally don't issue warnings about it.

Things to Remember

- `noexcept` is part of a function's interface, so callers may depend on it.
- `noexcept` functions are more optimizable than non-`noexcept` functions.
- `noexcept` is particularly valuable for the move operations and for `swap`.
- Most functions are exception-neutral rather than `noexcept`.