not required to destroy any objects whose lifetime ends between the **throw** and the function entry. Skipping this unwinding allows the compiler to eliminate otherwise unused cleanup code, producing a smaller program. Moreover, when the compiler sees that a function has a nonthrowing exception specification, it can safely assume that no exceptions will be thrown when calling that function and so can eliminate other cleanup code associated with handling potentially thrown exceptions; see *Use Cases — Reducing object-code size* on page 1101.

Finally, explicit use of **noexcept** is not an entirely new code-elimination opportunity as the compiler could perform a similar analysis on, say, the body of an **inline** function (or, for smaller programs, on the compiled application during **link-time optimization**). However, because the **noexcept** specifier resides on the *declaration* of the function, that specifier is necessarily visible when the caller is compiled. As a result, explicit use of an exception specification simplifies the analysis a compiler would have to perform, making the potential optimization more viable when separately compiling each individual **translation unit** and, hence, more likely, but see *Potential Pitfalls — Overly strong contracts guarantees* on page 1112 and *Potential Pitfalls — Unrealizable runtime performance benefits* on page 1134.

Use Cases

Declaring nonthrowing move operations

The most common algorithmic benefits of the **noexcept** feature accrue to types having move and swap operations that are guaranteed not to throw. Operations such as resizing an `std::vector`, for example, can use **move construction** instead of **copy construction** to transfer elements from a smaller memory buffer to a larger one without concern that an exception will occur in the middle (e.g., due to potential dynamic memory allocation) and thus leave the vector in a half-moved state. It therefore behooves us to consider whether our classes can have such nonthrowing move and swap operations whenever runtime performance matters and annotate them with **noexcept** where applicable.

The first question we must ask ourselves when considering the move operations of a new class is whether the class can benefit from having a **move constructor** or **move-assignment operator**. A class that does not allocate resources seldom needs move operations that are distinct from its **copy operations**. If resources are managed by one or more **data members** or base classes, the defaulted move operations are often sufficient. Note that any implicitly defaulted move operation will be suppressed by a user-declared copy operation; e.g., a user-declared **copy constructor** will suppress an implicit move constructor. See Section 1.1. "Defaulted Functions" on page 33.

A *user-provided* move operation will, by default, be **noexcept(false)**; it can and should be declared with the **noexcept** specifier whenever it does not invoke any throwing operation during the move. Let's, for example, define a **smart pointer** class, `CloningPtr`, that owns its pointed-to object and whose copy constructor and **copy-assignment operator** copy the owned object; two `CloningPtr` objects will never point to the same object:

```
template <typename T>
class CloningPtr
```