In the example above, the call to `n2` when `N` is `1` selects the leaf case (nontemplate) version and does not recursively instantiate the template version of `n2`.

## Potential Pitfalls

### Negative impacts on abstraction and insulation

If a library function provides an **abstract interface**, the user needs to read and understand only the function's declaration and its documentation. Except when maintaining the library itself, the function's implementation details are unimportant.

If a program **insulates** a library user from the library's implementation by placing the implementation code in a separate translation unit, **compile-time coupling** between library code and client code is reduced. A library that does not include function implementations in its header files can be rebuilt to provide updates without needing to recompile clients; only a relink is needed. Compilation times for client code are minimized by not needing to recompile library source code within header files.

Deduced function return types interfere with both abstraction and **insulation** and thus with the development of large-scale, comprehensible software. Because the return type cannot be determined without its implementation being visible to the compiler, publicly visible functions having deduced return types cannot be insulated; they must necessarily appear in a header file as **`inline`** functions or function templates, thereby being recompiled for every client translation unit. In this regard, a function with deduced return type is no different than any other **`inline`** function or function template. What is new, however, is its impact on abstraction: To fully understand a function's interface — including its return type — the user must read its implementation.

To mitigate the loss of abstraction from deduced return types, the function author can carefully document the expected properties of the returned object, even in the absence of a specific concrete type. Interestingly, understanding the return value's *properties*, not merely its *type*, might yield a resulting function that is *more* abstract than one for which a known type had been specified.

### Reduced clarity

Not having the return type of a function visible in its declaration can reduce the clarity of a program. Deduced return types work best when they appear on tiny function definitions, so that the determinative **`return`** statement is easily visible. Functions having deduced return types are also well suited for situations where the particulars of a return type are not especially useful, as in the case of iterator types associated with containers.