without updating the type of the iteration variable `i` in lockstep, the loop might silently[1] become infinite.[2]

Had **`decltype`**`(packet.checksumLength())` been used to express the type of `i`, the types would have remained consistent, and the ensuing defect would naturally have been avoided:

```
// ...
for (decltype(data.checksumLength()) i = 0; i < data.checksumLength(); ++i)
// ...
```

Creating an auxiliary variable of generic type

Consider the task of implementing a generic `loggedSum` function template that returns the sum of two arbitrary objects, `a` and `b`, after logging both the operands and the result value, e.g., for debugging or monitoring purposes. To avoid computing the possibly expensive sum twice, we decide to create an auxiliary function-scope variable, `result`. Since the type of the sum depends on both `a` and `b`, we can use **`decltype`**`(a + b)` to infer the type for both the trailing return type of the function (see Section 1.1."Trailing Return" on page 124) and the auxiliary variable:

```
template <typename A, typename B>
auto loggedSum(const A& a, const B& b)
    -> decltype(a + b)                // (1) exploiting trailing return types
{
    decltype(a + b) result = a + b;    // (2) auxiliary generic variable
    LOG_TRACE << a << " + " << b << " = " << result;
    return result;
}
```

Using **`decltype`**`(a + b)` as a return type is significantly different from relying on automatic **`return-type deduction`**; see Section 2.1."**`auto`** ~~Variables~~" on page ~~195~~. Note that this particular use involves significant repetition of the expression `a+b`. See *Annoyances — Mechanical repetition of expressions might be required* on page 31 for a discussion of ways in which such repetition might be avoided.

Determining the validity of a generic expression

In the context of generic-library development, **`decltype`** can be used in conjunction with **SFINAE** ("Substitution Failure Is Not An Error") to validate an expression involving a template parameter.

[1] As of this writing, neither GCC 11.2 (c. 2021) nor Clang 12.0.0 (c. 2021) provide a warning (using `-Wall`, `-Wextra`, and `-Wpedantic`) for the comparison between `std::uint8_t` and `std::uint16_t` — even if (1) the value returned by `checksumLength` does not fit in a 8-bit integer, and (2) the body of the function is visible to the compiler. Decorating `checksumLength` with **`constexpr`** causes `clang++` to issue a warning, which is clearly not a general solution.

[2] The loop variable is promoted to an **`unsigned int`** for comparison purposes but wraps to 0 whenever its value prior to being incremented is 255.