As an academically interesting example of this ~~practical security~~ problem, suppose we want to write a compile-time function in C++ to compute the **Collatz length** of an arbitrary positive integer and generate a compilation error if any intermediate calculation would result in signed integer overflow.

First let's take a step back to understand what we mean by Collatz length. Suppose we have a function, `cf`, that takes a positive **`int`**, `n`, and for even `n` returns `n/2` and for odd `n` returns `3n+1`:

```
int cf(int n) { return n % 2 ? 3 * n + 1 : n / 2; }  // Collatz function
```

Given a positive integer, `n`, the **Collatz sequence**, `cs(n)`, is defined as the sequence of integers generated by repeated application of the **Collatz function** — e.g., `cs(1) = { 4, 2, 1, 4, 2, 1, 4, ... };`, `cs(3) = { 10, 5, 16, 8, 4, 2, 1, 4, ... }`, and so on. A classic but as yet unproven conjecture in mathematics states that, for every positive integer, $n$, the Collatz sequence for $n$ will eventually reach 1. The Collatz length of the positive integer $n$ is the number of iterations of the Collatz function needed to reach 1, starting from $n$. Note that the Collatz length for `n = 1` is 0.

This example showcases the need for a **`constexpr`** variable in that its initializer is required to be a constant expression, ensuring that the evaluation of a **`constexpr`** function occurs at compile time. Again, to avoid distractions related to implementing more complex functionality within the limitations of C++11 **`constexpr`** functions, we will make use of the relaxed restrictions of C++14; see Section 2.1."**`constexpr`** Functions" on page 257:

```
constexpr int collatzLength(long long number)
    // Return the length of the Collatz sequence of the specified number. The
    // behavior is undefined unless each intermediate sequence member can be
    // expressed as a long long and number > 0.
{
    int length = 0;         // collatzLength(1) is 0.

    while (number > 1)      // The current value of number is not 1.
    {
        ++length;           // Keep track of the length of the sequence so far.

        if (number % 2)     // if the current number is odd
        {
            number = 3 * number + 1;    // advance from odd sequence value
        }
        else
        {
            number /= 2;                // advance from even sequence value
        }
    }

    return length;
}
```