```
    // In this case, the second argument will be partitioned as the first
    // type in the sequence and the possibly empty remainder of the
    // TypeList. The compile-time value of the base class will be either the
    // same as or one greater than the value accumulated in the TypeList so
    // far, depending on whether the first element is the same as the one
    // supplied as the first type to Count.

static_assert(Count<int, TypeList<int, char, int, bool>>::value == 2, "");
```

Notice that we made use of a C++11 **parameter pack**, `Tail...` (see Section 2.1."Variadic Templates" on page 873), in the implementation of the simple template specialization to package up and pass along any remaining types.

The C++11 restrictions encourage both somewhat rarefied metaprogramming-related knowledge and a *recursive* implementation that can be compile-time intensive in practice. For a more efficient C++11 version of `Count`, see ***constexpr*** *type list* `Count` *algorithm* in *Appendix — Optimized C++11 example algorithms* on page 966. By exploiting C++14's relaxed **constexpr** rules, a simpler and typically more compile-time friendly *imperative* solution can be realized:

```
template <typename X, typename... Ts>
constexpr int count()
{
    bool matches[sizeof...(Ts)] = { std::is_same<X, Ts>::value... };
        // Create a corresponding array of bits where 1 indicates sameness.

    int result = 0;
    for (bool m : matches)  // (C++11) range-based for loop
    {
        result += m;        // Add up 1 bits in the array.
    }

    return result;  // Return the accumulated number of matches.
}
```

The implementation above — though more efficient and comprehensible — will require some initial learning for those unfamiliar with C++ variadic templates. The general idea here is to use **pack expansion** in a nonrecursive manner (see Section 2.1."Variadic Templates" on page 873) to initialize the `matches` array with a sequence of zeros and ones (representing, respectively, mismatches and matches between `X` and a type in the `Ts...` pack) and then iterate over the array to accumulate the number of ones as the final `result`. This **constexpr**-based solution is both easier to understand and typically faster to compile.[6]

[6]For a type list containing 1,024 types, the imperative (C++14) solution compiles about twice as fast on GCC 11.2 (c. 2021) and roughly 2.6 times faster on Clang 12.0.1 (c. 2021).