Optimized metaprogramming algorithms

C++14's relaxed **constexpr** restrictions enable the use of modifiable local **variables** and **imperative** language constructs for **metaprogramming** tasks that were historically often implemented by using Byzantine recursive **template instantiations**, notorious for their voracious consumption of compilation time.

Consider, as a simple example, the task of counting the number of occurrences of a given type inside a **type list** represented here as a variadic template (see Section 2.1."Variadic Templates" on page 873) that can be instantiated using a **variable**-length sequence of arbitrary C++ types:

```
template <typename...> struct TypeList { };
    // empty variadic template instantiable with arbitrary C++ type sequence
```

Explicit instantiations of this variadic template could be used to create objects:

```
TypeList<>                  emptyList;
TypeList<int>               listOfOneInt;
TypeList<int, long, double> listOfThreeIntLongDouble;
```

A naive C++11-compliant implementation of a **metafunction** `Count`, used to ascertain the number of times a given C++ type was used when creating an instance of the `TypeList` template (above), would usually make recursive use of involved **partial class template specialization** to satisfy the single-return-**statement** requirements:

```
#include <type_traits>  // std::integral_constant, std::is_same

template <typename X, typename List> struct Count;
    // general template used to characterize the interface for the Count
    // metafunction

    // Note that this general template is an incomplete type.

template <typename X>
struct Count<X, TypeList<>> : std::integral_constant<int, 0> { };
    // partial class template specialization of the general Count template
    // (derived from the integral-constant type representing a compile-time
    // 0), used to represent the base case for the recursion --- i.e., when
    // the supplied TypeList is empty

    // The payload (i.e., the enumerated value member of the base class)
    // representing the number of elements of type X in the list is 0.

template <typename X, typename Head, typename... Tail>
struct Count<X, TypeList<Head, Tail...>>
    : std::integral_constant<int,
        std::is_same<X, Head>::value + Count<X, TypeList<Tail...>>::value> { };
    // partial class template specialization of the general Count template
    // for when the supplied list is not empty
```