See Also

- "**constexpr** Functions" (§2.1, p. 257) describes fundamentals of compile-time function evaluation.
- "**constexpr** Variables" (§2.1, p. 302) introduces variables usable as constant expressions.
- "Variadic Templates" (§2.1, p. 873) introduces a feature that allows templates to accept an arbitrary number of template arguments.

Further Reading

- Scott Meyers advocates for aggressive use of `constexpr` in **meyers15b**, "Item 15: Use constexpr whenever possible," pp. 97–103.

Appendix

Optimized C++11 example algorithms

Recursive Fibonacci Even with the restrictions imposed by C++11, we can write a more efficient recursive algorithm to calculate the nth Fibonacci number:

```
#include <utility>  // std::pair

constexpr std::pair<long long, long long> fib11NextFibs(
    const std::pair<long long, long long> prev,  // last two calculations
    int count)                                    // remaining steps
{
    return (count == 0) ? prev : fib11NextFibs(
        std::pair<long long, long long>(prev.second,
                                        prev.first + prev.second),
        count - 1);
}
```

```
constexpr long long fib11Optimized(long long n)
{
    return fib11NextFibs(
        std::pair<long long, long long>(0, 1), // first two numbers
        n                                      // number of steps
    ).second;
}
```

constexpr type list Count algorithm As with the `fib11Optimized` example, providing a more efficient version of the `Count` algorithm in C++11 is also possible, by accumulating the final result through recursive **constexpr** function invocations:

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

template <typename>
constexpr int count11Optimized() { return 0; }
    // Base case: always return 0.

template <typename X, typename Head, typename... Tail>
constexpr int count11Optimized()
    // Recursive case: compare the desired type (X) and the first type in
    // the list (Head) for equality, turn the result of the comparison
    // into either 1 (equal) or 0 (not equal), and recurse with the rest
    // of the type list (Tail...).
{
    return std::is_same<X, Head>::value + count11Optimized<X, Tail...>();
}
```

This algorithm can be optimized even further in C++11 by using a technique similar to the one shown for the iterative C++14 implementation. By leveraging an `std::array` as compile-time storage for bits where `1` indicates equality between types, we can compute the final result with a fixed number of template instantiations:

```
#include <array>        // std::array
#include <type_traits>  // std::is_same

template <int N>
constexpr int count11VeryOptimizedImpl(
    const std::array<bool, N>& bits,  // storage for "type sameness" bits
    int i)                            // current array index
{
    return i < N
        ? bits[i] + count11VeryOptimizedImpl<N>(bits, i + 1)
            // Recursively read every element from the bits array and
            // accumulate into a final result.
        : 0;
}
```