The implementation follows a head-and-tail recursion that is typically used for C++ variadic function templates. The first overload of `print` has no parameters and simply outputs a newline to the console. The second overload does the bulk of the work. It takes one or more arguments, prints the first, and recursively calls `print` to print the rest. In the limit, `print` is called with no arguments, and the first definition kicks in, outputting the line terminator and also ending the recursion.

A variadic function's smallest number of allowed arguments does not have to be zero, and it is free to follow many other recursion patterns. For example, suppose we want to define a variadic function `isOneOf` that returns **`true`** if and only if its first argument is equal to one of the subsequent arguments. Calls to such a function are sensible for two or more arguments:

```
template <typename T1, typename T2>     // normal template function
bool isOneOf(const T1& a, const T2& b)  // two-parameter version
{
  return a == b;
}

template <typename T1, typename T2, typename... Ts>      // two or more arguments
bool isOneOf(const T1& a, const T2& b, const Ts&... xs)  // all by const&
{
  return a == b || isOneOf(a, xs...);                    // compare, recurse
}
```

Again, the implementation uses two definitions in a pseudorecursive setup but in a slightly different stance. The first definition handles two items and also stops recursion. The second version takes three or more arguments, handles the first two, and issues the recursive call only if the comparison yields **`false`**.

Let's take a look at a few uses of `isOneOf`:

```
#include <string>  // std::string

int a = 42;
bool b1 = isOneOf(a, 1, 42, 4);   // b1 is true.
bool b2 = isOneOf(a, 1, 2, 3);    // b2 is false.
bool b3 = isOneOf(a, 1, "two");   // Error, can't compare int with const char*
std::string s = "Hi";
bool b4 = isOneOf(s, "Hi", "a");  // b4 is true.
bool b5 = isOneOf(s);  // Error, no overload takes fewer than two parameters.
```

Processing variadic arguments in order

Let's now consider two possible implementations of the variadic string concatenation function `concat`, introduced in *Description* on page 873: one using a recursive approach, and the other taking advantage of braced initialization (see Section 2.1."Braced Init" on page 215) to avoid recursion.