Using conventional function templates, we can drastically reduce the volume of source code required, albeit with some manageable increase in implementation complexity.

Each of the $N + 1$ templates can be written to accept any combination of its M arguments ($0 \leq M \leq N$) such that each parameter will independently bind to a **const char***, an `std::string`, or a **char** with no unnecessary conversions or extra copies at run time. Of the exponentially many possible `concat` **template instantiations**, the compiler generates — on demand — only those overloads that are actually invoked.

With the introduction of variadic templates in C++11, we are now able to represent variadic functions such as `add` or `concat` with just a single template that expands automatically to accept any number of arguments of any appropriate types — all by, say, **const lvalue reference**:

```
template <typename... Ts>
std::string concat(const Ts&...);
    // Return a string that is the concatenation of a sequence of zero or
    // more character or string arguments --- each of a potentially distinct
    // C++ type --- passed by const lvalue reference.
```

A **variadic function template** will typically be implemented with **recursion** to the same function with fewer parameters. Such function templates will typically be accompanied by an overload (templated or not) that implements the lower limit, in our case, the overload having exactly zero parameters:

```
std::string concat();
    // Return an empty string ("") of length 0.
```

The nontemplate overload above **declares** `concat` taking no parameters. Importantly, this overload will be preferred for calls to `concat` having no arguments because ~~the nontemplate function is~~ a better match than the variadic **declaration**, even though the variadic declaration would also accept zero arguments.

Having to write just two overloads to support any number of arguments has clear advantages over writing dozens of overloaded templates: (1) there is no hard-coded limit on argument count, and (2) the source is dramatically smaller, more regular, and easier to maintain and extend — e.g., it would be easy to add support for efficiently passing by **forwarding reference** (see Section 2.1."Forwarding References" on page 377). A second-order effect should be noted as well. The costs of defining variadic functions with C++03 technology are large enough to discourage such an approach in the first place, unless overwhelming efficiency motivation exists; with C++11, the low cost of defining variadics often makes them the simpler, better, and more efficient choice altogether. We return to the `concat` function template and provide a complete implementation later; see *Use Cases — Processing variadic arguments in order* on page 926.

Variadic *class* templates are another important motivating use case for this language feature.

A tuple is a generalization of `std::pair` that, instead of comprising just two objects, can store an arbitrary number of objects of heterogeneous types: