```
template <typename T> class C3;    // (1) primary declaration; C3 not variadic

template <typename K, typename V>  // (2a) Specialize C3 for all Maps, C++03 style.
class C3<Map<K, V>>;
```

Variadics offer a terser, more flexible alternative:

```
template <typename... Ts>  // (2b) Specialize C3 for all Maps, variadic style.
class C3<Map<Ts...>>;
    // Note: Map works with pack expansion even though it's not variadic,
    // but Ts... must comprise the number and types of arguments appropriate
    // for instantiating one.
```

The most important advantage of (2b) over (2a) is flexibility. In maintenance, `Map` might acquire additional template arguments, such as a comparison functor and an allocator. Changing `Map`'s arguments requires surgery on `C3`'s specialization (2a), whereas (2b) will continue to work unchanged because `Map<Ts...>` will accommodate any additional template parameters `Map` might accummulate.

An application must use either the nonvariadic (2a) or the variadic (2b) partial specialization but not both. If both are present, (2a) is always preferred because an exact match is always more specialized than one that deduces a parameter pack.

Variadic alias templates

Alias templates (since C++11) are a new way to associate a name with a family of types without needing to define forwarding glue code. For full details on the topic, see Section 1.1. "**using** Aliases" on page 133. Here, we focus on the applicability of template parameter packs to alias templates.

Consider, for example, the `Tuple` artifact that can store an arbitrary number of objects of heterogeneous types:

```
template <typename... Ts> class Tuple;  // Declare Tuple.
```

Suppose we want to build a simple abstraction on top of `Tuple` — a "named tuple" that has an `std::string` as its first element, followed by anything a `Tuple` can store:

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

template <typename... Ts>
using NamedTuple = Tuple<std::string, Ts...>;
    // Introduce alias for Tuple of std::string and anything.
```

In general, alias templates take template parameter packs following the same rules as primary class template declarations: An alias template is allowed to take at most one template parameter pack in the last position. Alias templates do not support specialization or partial specialization.