```
template <typename... Ts, typename... Us, typename T>
int process(Ts..., Us..., T);
    // Ill-formed declaration: Us must be empty in every possible call.
int x = process<int, double>(1, 2.5, 3);
    // Ts=<int, double>, Us=<>, T=int
```

In virtually all cases, such code reflects a misplaced expectation; an always-empty parameter pack has no reason to exist in the first place.

Compiler limits on the number of arguments

The C++ Standard recommends that compilers support at least 1,024 arguments in a variadic template instantiation. Although this limit seems generous, real-world code might bump up against it, especially in conjunction with generated code or combinatorial uses.

This limit might lead to a lack of portability in production — for example, code that works with one compiler but fails with another. Suppose, for example, we ~~define~~ `Variant` that carries all possible types that can be serialized in a large application:

```
typedef Variant<
    char,
    signed char,
    unsigned char,
    short,
    unsigned short
    // ...                  (more built-in and user-defined types)
>
WireData;
```

We release this code to production and then, at some later date, clients find it fails to build on some platforms, leading to a need to reengineer the entire solution to provide full cross-platform support.

Annoyances

Unusable functions

Before variadics, any properly defined template function could be called by using explicit template argument specification, type deduction, or a combination thereof. Now it is possible to define variadic function templates that pass compilation but are impossible to call (either by using explicit instantiation, argument deduction, or both). Such unusable functions could cause confusion and frustration. Consider, for example, a few function templates, none of which take any function parameters: