Expansion is rigid and requires verbose support code

There are only two syntactic constructs that apply to parameter packs: **`sizeof...`** and expansion via `...`. The latter underlies virtually all treatment of variadics and, as discussed, requires handwritten support classes or functions as scaffolding toward building a somewhat involved recursion-based pattern.

There is no expansion in an expression context, so it is not possible to write functions such as `print` in a concise, single-definition manner; see *Use Cases — Generic variadic functions* on page 925. In particular, expressions are not expansion contexts~~, so the following code will not work~~:

```
#include <iostream>  // std::cout, std::ostream, std::endl

template <typename... Ts>
std::ostream& print(const Ts&... vs)
{
    std::cout << vs...;             // Error, invalid expansion
    return std::cout << std::endl;
}
```

Linear search for everything

One common issue with parameter packs is the difficulty of accessing elements in an indexed manner. Getting to the `n`th element of a pack is a linear search operation by necessity, which makes certain uses awkward and potentially time-consuming during compilation. Refer to the implementation of `destroyLog` in *Use Cases — Variant types* on page 937 as an example.

See Also

- "Braced Init" (§2.1, p. 215) illustrates one of the expansion contexts for function parameter packs.
- "Forwarding References" (§2.1, p. 377) describes a feature used in conjunction with **variadic function templates** to achieve perfect forwarding.
- "Lambdas" (§2.1, p. 573) introduces a feature that supports pack expansion in its capture list.