Parameter packs cannot be used unexpanded

As discussed in *Description — Pack expansion* on page 908, the name of a parameter pack cannot appear on its own in a correct C++ program; the only way to use a parameter pack is as part of an expansion by using `...` or **`sizeof`**. Such behavior is unlike types, template names, or values.

It is impossible to pass parameter packs around or to give them alternative names (as is possible with types by means of **`typedef`** and **`using`** and with values by means of references). Consequently, it is also impossible to ~~define~~ them as "return" values for metafunctions following conventions such as `::type` and `::value` that are commonly used in the `<type_traits>` standard header.

Consider, for example, sorting a type parameter pack by size. This simple task is not possible without a few helper types because there is no way to return the sorted pack. One necessary helper would be a typelist:

```
template <typename...> struct Typelist { };
```

With this helper type in hand, it is possible to encapsulate parameter packs, give them alternate names, and so on — in short, give parameter packs the same maneuverability that C++ types have:

```
typedef Typelist<short, int, long, float, double, long double> Numbers;
    // can be used to give a pack an alternate name

template <typename L>
struct SortBySize
{
    using type = Typelist< /*...*/ >;  // computed sorted-by-size version of
                                       // the Typelist L
};

typedef SortBySize<Numbers>::type SortedNumbers;
    // can be used to "return" a pack from a metafunction
```

Currently no `Typelist` facility has been standardized. An active proposal[26] introduces `parameter_pack` along the same lines as `Typelist` above. Meanwhile, compiler vendors have attempted to work around the problem in nonstandard ways.[27] A related proposal[28] defines `std::bases` and `std::direct_bases` but has, at the time of writing, been rejected.

[26] **spertus13**

[27] ~~GNU~~ defines the nonstandard primitives `std::tr2::__direct_bases` and `std::tr2::__bases`. The first yields a list of all direct bases of a given class, and the second yields the transitive closure of all bases of a class, including the indirect ones. To make these artifacts possible, ~~GNU~~ defines and uses a helper `__reflection_typelist` class template similar to `Typelist` above.

[28] **spertus09**