```
#include <utility>  // std::forward

template <typename T, typename... ARGS>
T factory1(ARGS&&... args)
{
    return T(std::forward<ARGS>(args)...);  // direct initialization
}

template <typename T, typename... ARGS>
T factory2(ARGS&&... args)
{
    return T{std::forward<ARGS>(args)...};  // direct list initialization
}

template <typename T, typename... ARGS>
T factory3(ARGS&&... args)
{
    return  {std::forward<ARGS>(args)...};  // copy list initialization
}
```

All three factory functions are defined using **perfect forwarding** (see Section 2.1."Forwarding References" on page 377) but support different subsets of C++ types and might interpret their arguments differently.

`factory1` returns a value created by direct initialization but, because it uses parentheses, cannot return an aggregate unless, as a special case, the `args` list is empty or contains exactly one argument of the same type `T` or one convertible to `T`; otherwise, the attempt to construct the return value will result in a compilation error.[6]

`factory2` returns an object created by direct list initialization. Hence, `factory2` supports the same types as `factory1`, plus aggregates. However, due to the use of braced initialization, `factory2` will reject any types in `ARGS` that require narrowing conversion when passed to the constructor (or to initialize the aggregate member) of the return value. Also, if the supplied arguments can be converted into a homogeneous `std::initializer_list` that matches a constructor for `T`, then that constructor will be selected, rather than the constructor best matching that list of arguments.

`factory3` behaves the same as `factory2` except that it uses copy list initialization and will thus also produce a compile error if the selected constructor or conversion operator for the return value is declared as **`explicit`**.

There is no one true form of initialization that works best in all circumstances for such a factory function, and library developers must choose and document in their contract the form that best suits their needs. Note that the Standard Library runs into this same problem when implementing factory functions like `std::make_shared` or the `emplace` function of any container. The Standard Library consistently chooses parentheses initialization like

[6] Note that C++20 will allow aggregates to be initialized with parentheses as well as with braces, which will result in this form being accepted for aggregates as well.