```
#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.

`function1` 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]

`function2` returns an object created by direct list initialization. Hence, `function2` supports the same types as `function1`, plus aggregates. However, due to the use of braced initialization, `function2` 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.

`function3` behaves the same as `function2` 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.

~~function~~1 in the previous code example, so these functions do not work for aggregates prior to C++20.

Uniform member initialization in generic code

With the addition of general braced initialization to C++11, class authors should consider whether constructors should use *direct* initialization or *direct list* initialization to initialize their bases and members. Note that since copy initialization and copy list initialization are not options, whether or not the constructor for a given base or member is **`explicit`** will never be a concern.

Prior to C++11, writing code that initialized aggregate subobjects, including arrays, with a set of data in the constructor's member initializer list was not really possible. We could only *default*-initialize, *value*-initialize, or *direct*-initialize from another aggregate of the same type.

Starting with C++11, we are able to initialize aggregate members with a list of values, using aggregate initialization in place of direct list initialization for members that are aggregates:

```
struct S
{
    int         i;
    std::string str;
};

class C
{
    int j;
    int a[3];
    S   s;

public:
    C(int x, int y, int z, int n, const std::string t)
    : j(0)
    , a{ x, y, z }  // ill formed in C++03, OK in C++11
    , s{ n, t }     // ill formed in C++03, OK in C++11
    {
    }
};
```

Note that as the initializer for `C.j` shows in the code example above, there is no requirement to consistently use either braces or parentheses for all member initializers.

As with the case of factory functions, the class author must make a choice for constructors between adding support for initializing aggregates versus narrowing conversion being ill formed. As mentioned earlier, since member initialization supports only *direct* list initialization, there is never a concern regarding **`explicit`** conversions in this context: