need not even allocate space for `unused` on the stack, let alone initialize its elements.[14] Note that the pack expansion always calls `extent` with exactly one argument, so there is no need to define `extent`'s recursive variadic overload or parameterless base-case overload.

The technique used to compute `resultLen` is applied again to append each element of `args` — be it a **`char`**, a **`const char`**`*`, or an `std::string` — to the end of `result`. Because the pack expansion context is an initializer list, the side effects of the expressions in the pack expansion are guaranteed to occur in order, unlike, say, a function argument list, which has no such ordering guarantee. Note that a pack expansion could be used to initialize a **temporary object** of type `std::initializer_list<bool>` to yield the same result by replacing `{ bool unused[] = { ... }; }` with `(void) initializer_list<bool>{ ... };` for the two uses of the idiom; see Section 2.1."`initializer_list`" on page 553.

Object factories

Suppose we want to define a generic factory function — a function able to create an instance of any given type by calling one of its constructors. Object factories[15,16] allow libraries and applications to centrally control object creation for a variety of reasons: using special memory allocation, tracing and logging, benchmarking, object pooling, late binding, deserialization, interning, and more.

The challenge in defining a ~~generic object factory~~ is that the type to be created (and therefore its constructors) is not known at the time of writing the factory. That's why C++03 object factories typically offer only default object construction, forcing clients to awkwardly use two-phase initialization, first to create an empty object and then to put it in a meaningful state.

Writing a generic function that can transparently forward calls to another function (**perfect forwarding**) has been a long-standing challenge in C++03. An important part of the puzzle is making the forwarding function generic in the number of arguments, which is where variadic templates help in conjunction with forwarding references (see Section 2.1. "Forwarding References" on page 377):

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

void log(const char* message);               // logging function

template <typename Product, typename... Ts>   // type to be created and params
Product factory(Ts&&... xs)                  // call by forwarding reference
{
    log("factory(): Creating a new object"); // Do some logging.
    return Product(std::forward<Ts>(xs)...);  // Forward arguments to ctor.
}
```

[14]Experiments with both Clang 12.0.0 (c. 2021) and GCC 11.1 (c. 2021) show that the `unused` array is entirely optimized away at optimization level `-O2`, even if the `{` and `}` braces surrounding the definition of `unused` are omitted.

[15]**gamma95**, Chapter 3, section "Factory Method," pp. 107–115

[16]**alexandrescu01**, Chapter 8, "Object Factories," pp. 197–218