Doc. no.:	P0922R0
Date:	2018-02-10
Reply to:	Titus Winters
Audience:	EWG

This paper is a follow-up on conversations from the ABQ meeting and reflector discussion in late 2017 identifying areas that LEWG would like to see EWG focus on. I've attempted to keep notes on those discussions and extract relevant/unresolved discussion points from the reflector - any omissions are unintentional. I encourage other committee members and interested parties to produce similar wishlists.

The following list is unstructured and ill-thought out: the fundamental problem of inviting librarians to do the work of language-designers is that we are (or at least I am) outside of our area of expertise. Attention should be paid to the problems presented - any suggestion of how to resolve these is obviously half-baked at best. But as we all know - the best way to get a good answer to a question is to provide a wrong one and wait for the outraged corrections.

ADL

Argument-dependent Lookup is currently very difficult for librarians. Anecdotally, Google C++ team investigations suggest that the number of functions (other than overloaded operators) being called via ADL unintentionally, or outside the API design for that function, outweighs the number of correct uses of ADL by at least a factor of 10x. ADL makes library maintenance meaningfully harder - namespace changes cause build breaks when users are unexpectedly / unintentionally relying on ADL.

As a straw-man: introduce some mechanism to say "this API is ADL-enabled" and tag standard APIs (swap, non-member operators) in that fashion. Begin to warn on use of ADL to call non-ADL APIs, and eventually turn off old-style/accidental ADL. We could conceivably do this in 10 years.

In private communication, Richard Smith suggested that it might be valuable to find an ADL alternative that doesn't globally reserve a name. I'm interested in that idea, but not convinced it is an absolute necessity: the APIs I've seen that are designed with ADL in mind are pretty clear (swap, operators, and similar non-member/supplemental APIs). Globally reserving things like "swap" or "parseFlag" (a Google-internal API designed with ADL usage in mind) doesn't seem like an unreasonable thing - although it might be nice to avoid if possible.

Other ideas that have been kicked around: only consider friend functions. This would mean that there is explicit opt-in somewhere, and the name does not wind up globally reserved.

Types, lifetimes, and side-effects

If it were possible to explicitly state whether a type does or does not have external side effects, we could solve a couple related classes of user bugs. Consider two very different styles of type design: RAII types, and regular types.

In the case of RAII types, like `std::scoped_lock`, there is a common class of user error stemming from constructing these as a temporary.

```
std::scoped_lock{my_mutex};
```
vs.
```
std::scoped_lock l{my_mutex};
```

If we had a mechanism to express classes of side-effect (or appropriateness to operate on temporaries), we could more fundamentally quash this class of (common, hard to spot) bugs.

On the flip side, identifying types that are regular/value types could help identify places where temporaries are being used inappropriately. Consider

```
std::string GetString() { return {}; }
GetString() = "Hello";

int GetInt() { return 17; }
GetInt() = 42;
```

In the above example, assignment to `GetInt()` fails, but assignment to `GetString()` succeeds - and is almost certainly indicative of a bug in the code or the mental model for the programmer. There has been some recent discussion on the reflectors about reference-qualifying `operator=` - this might be sufficient. If EWG believes that **is** sufficient, LEWG would like to know so that we can investigate deployment of that as an idiom.

More Fun With Temporaries

With the introduction of `std::string_view` we have made object-lifetime bugs more common, at least for some classes of user. Invariably, the suggestion becomes “Why don’t we disallow construction from temporaries?”

```
class string_view {
 public:
  string_view(const string&);
  string_view(string&&) = delete;
  …
};
```

The problem is that this disallows a large and common class of usages:

```
void f(std::string_view);
std::string GetString();

void call() {
  f(GetString());
}
```

This is clearly safe and is in fact one of the primary purposes of `string_view` - provide a vocabulary type for string-like things as (non-sink) parameters. That said, it is still clearly the case that it is too easy to do the following:

```
void f() {
  std::string_view sv = GetString();
}
```

I think the ultimate example in this type of problem is `FindWithDefault` - a seemingly innocuous helper for associative containers that takes a map, a key, and a default value and returns a constant reference to the specified element or the default if the key does not exist. The problem here is similar:

```
// dangerous - the temporary passed to FindWithDefault is destructed
// at the end of the statement, the reference dangles if the key is not found.
const std::string& val = FindWithDefault(my_map, key, "default value");

// perfectly safe
UseString(FindWithDefault(my_map, key, "default value"));
```

Passing a temporary as the default is dangerous when the result is stored, but perfectly safe within the confines of a single statement. Users similarly suggest deleting the rvalue-ref overload for `FindWithDefault`, losing a large class of valid use cases.

As it stands, the standard provides no way for the type system to identify these types of lifetime concerns. Static analysis is being employed to identify unsafe uses and warn users, but there is no overall solution without deeper language intervention.

User-defined conversions to standard types

User-defined conversions to standard types interact poorly with the standard (ever) changing its idioms for parameter passing.

Consider `string_view` vs. `const char*` + `const string&` overloads - we would like to have only `string_view` parameters for string-accepting APIs. This is especially important because `string_view` works best when it's used consistently at the lowest-levels of a codebase (see http://abseil.io/tips/1). Consistent use of `string_view` by the standard would also provide good examples to model. However, because of user-code that has implicit conversion to `std::string` (only), we cannot do so without breaking such callers.

It would be nice if the standard were able to model the behavior we suggest for users.

ABI

At some point in the future it is likely that the language willforce another ABI break on the world. What will it take for that to be the last one? Or should we focus instead on making ABI breaks easier to handle?

Macros

What will it take for us to stop relying on macros in 10 years? Currently I encounter macros in the following cases:

- Stamping out near-duplicate chunks of code, textually.
- Need to access the filename/linenumber of a "caller"
- Desire to affect control flow in new and exciting ways
- Need to access the name of a declaration
- stringifcation of arguments
- Deferred evaluation of parameters/parameter-like expressions

Much of this is handled by SG7 - are we sufficiently certain that we've covered all of the current use-cases?

Acknowledgements

Thank you to everyone that participated in the reflector discussion in December 2017. Particular thanks to Richard Smith and David Jones for insightful comments on early drafts of this paper.