Parameter passing →
guaranteed unified initialization and unified value-setting

Document Number: **d0708 R0 – draft 30**
Date: 2021-11-08
Reply-to: Herb Sutter (hsutter@microsoft.com)

"How do I pass parameters?" and "how do I initialize variables?" empirically accounts for ~23**%** of the total rules in major C++ guideline literature, most of which would be eliminated by this proposal. This paper aims to:

Provide **declarative** parameter passing **consistently** for all parameters (templates and non-templates).

Reduce tedious boilerplate code, including to automate laborious rote **optimizations** currently done by hand.

Provide strong **initialization-before-use guarantees** uniformly for objects of all types (PODs and non-PODs).

Provide true **unified initialization** syntax and semantics and **unified value-setting** for `operator=`.

Contents

0 Overview 2
0.1 Background and motivation 2
0.2 Root cause: "How" to pass parameters, by idiom and convention 2
0.3 Design principles 4
0.4 Acknowledgments 4
1 "What, not how" parameter passing 5
1.1 "What": Directly declare intent 5
1.2 Definite first/last use (see also P1179) 8
1.3 Parameters 9
1.4 Range-`for` 19
1.5 Return values 20
1.6 Overloading 21
1.7 Segue: The `this` parameter 22
2 Generalizations: Guaranteed and unified initialization 23
2.1 `out` enables consistent `= uninitialized` 23
2.2 `out` is a named and unified { constructor, value-setter } 26
2.3 `= initializer` enables consistent and truly uniform initialization 27
2.4 Segue toward generalized `operator=`: Zero/multiple params 28
2.5 Generalized `operator=`, part 1: `in` "that" 29
2.6 Generalized `operator=`, part 2: `out` "`this`" 30
2.7 Together: Unified {copy,move}×{construction,assignment} 31
Appendix A: Implementation options for §1 33
A.1 Basic option 1: Static 34
A.2 Basic option 2: Dynamic 35
A.3 Current approach: Static for `forward`, dynamic for `in` and `out` 36
A.4 Multiple `in` and/or `forward` in the same definite last use 38
Bibliography 40

0 Overview

0.1 Background and motivation

C++ has perhaps the largest guidance and coding standard literature of any programming language. The bad news is that there is so much to teach; the good news is that our community has already documented what we think are the important things a C++ programmer has to know, in a way that can be measured and used to draw conclusions about how to improve the language.

I am performing a review of the C++ guidance literature to catalog and analyze all the rules they teach. Of the 638 rules I have catalogued and analyzed so far, ~16% are about "how do I pass parameters?" and another ~7% are about "how do I initialize variables?" (including ~2% of informational explanations such as "what if I don't have program-meaningful values when I have to declare a variable?" and "is = in a declaration an assignment?").

Most of those ~23% of all published C++ guidance would be eliminated by the features in this paper, so that we would not have to teach them anymore for new modern code. The resulting code will not only be simpler, but will be more robust and correct (by eliminating classes of errors), and at least as efficient (and often more efficient, by eliminating the need for useless work such as dead writes which are required today).

0.2 Root cause: "How" to pass parameters, by idiom and convention

In today's C++, we require the programmer to specify the mechanics of "how" to pass a parameter, and so "what" the parameter is intended to be used for is expressed only by indirect convention. This created a need to invent (eventually multiple kinds of) references, particularly for natural operator overloading.

Today, a function can take a parameter via any combination of 10 major options:

{ `const`, non-`const` } × { value, pointer, lvalue reference, rvalue reference, forwarding reference }

However, some of these are **meaningless**, so we have to teach not to use them, and why. For example:

- An "rvalue reference to `const`" parameter is meaningless, because it prevents moving.
- A "forwarding reference to `const`" parameter is probably meaningless, because it would forward only lvalue/rvalue-ness while treating the object as const, and since "rvalue reference to `const`" is meaningless its only remaining useful meaning would be the same as just declaring an lvalue reference to `const`.

Worse, some combinations are **ambiguous**, so every major compiler and platform of the past 30 years has repeatedly reinvented nonstandard annotations to supply the missing information. For example:

- An `X&` parameter is ambiguous about whether the parameter is:
  - an "inout" parameter,
  - an initialized "out" parameter to be filled in by value, or
  - an uninitialized "out" parameter to be filled in by construction.

  The usually correct guess is "inout" because that is the most common, but there is no way to express "out" which is important because the initial value should not be read.
- An `X*` parameter is ambiguous about whether the primary object being passed is:
  - the pointer itself (e.g., with C-style `FILE*` APIs the pointer itself is the value),

  - an `X` that only for { flexibility, optionality } happens to be passed by pointer (which in turn may or may not be safe to store to outlive the function call),
  - an initialized "out" parameter to be filled in by setting its value (usually assignment), or
  - an uninitialized "out" parameter to be filled in by construction.

  And so if the intent is one of the "out" cases we cannot enforce that the initial value should not be read.

- An `X**` parameter is ambiguous about whether it means:
  - an `X**` "in" parameter,
  - an `X*` "inout" parameter, or
  - an `X*` "out" parameter.

And, unfortunately, some combinations are useful but **difficult or excessively intricate to write**. For example:

- A "forwarding reference to `X`" parameter where `X` is a concrete type is difficult to write. The workaround is to write the function instead as a template whose type parameter is constrained to `X` using `requires` or `enable_if`.

- An "rvalue reference to `T`" parameter where `T` is a template parameter type is probably impossible to write in a strictly perfect way that includes using `std::move` in the body, but you can get very close by writing `T&&` (a forwarding reference), using `enable_if` or `requires` to constrain the type to reject deduced lvalue references, and in the body writing `std::forward<T>` instead of `std::move`.

The table at right summarizes most of the rules we repeatedly teach today.

	What we teach today: "How" mechanics
In	Pass by value for "cheap to copy/move" types (incl. builtin types) Otherwise, pass by **const X&** + **Overload** non-templated rvalue reference **X&&** + **std::move** once to optimize rvalues **except** if X must be a type parameter, write templated forwarding reference **X&&** + **enable_if/requires is_reference_v<X>** and **std::forward** instead **except** consider passing **X by value** if it's an "**in+copy**" parameter to a constructor
In-out	Pass by non-const **X&**
Out	Pass by non-const **X&** + nonstd annotations Can't distinguish from in-out in the language Can't enforce write-before-read or must-write
Move	Pass by non-templated rvalue reference **X&&** + **std::move** once **except** if X must be a type parameter, write templated forwarding reference **X&&** + **enable_if/requires !is_reference_v<X>** and **std::forward** instead
Forward	Pass by templated forwarding reference **T&&** + **std::forward** once **and** if we want only a concrete type X, add **enable_if/requires is_convertible_v<T,X>**

There's a lot wrong here. Most embarrassing of all is that **"in," the most common case, is the worst**: It's bad enough that for "in" we teach writing an overload set instead of a single function (never mind that the suggested overload doesn't even work for templates where `&&` means something else), and possibly overload a function template (even if the original function was not a template)… except possibly for constructors (not even mentioned here, but see [Sutter 2014])… but that's not all, because for a function with N "in" parameters that aren't cheap to copy, following the guidance requires writing 2^N overloads for each combination of lvalue and rvalue arguments… and so some experts just write a perfect forwarder instead even though that is very advanced and isn't even expressing "in" read-only-ness anymore… and we have failed.

This advice is essentially unteachable, which explains why it continues to be debated even among experts.

Note When move semantics were new, some trainers and style guides hoped that this would let us simplify the guidance to simply pass "in" objects by value always. That guidance definitely felt like a breath of fresh air compared to the complexity of teaching overloads and different kinds of references, and it did automatically move from rvalues, but it was flawed (led to usability and performance pitfalls) and it is not embraced by most guidance literature or the C++ standard library.

My CppCon 2014 talk [Sutter 2014] shows how complicated this gets even when seriously attempting to simplify the guidance. We're still teaching and debating how to pass C++ parameters; there's something wrong with that.

0.3 Design principles

Note These principles apply to all design efforts and aren't specific to this paper. Please steal and reuse.

The primary design goal is conceptual integrity [Brooks 1975], which means that the design is coherent and reliably does what the user expects it to do. Conceptual integrity's major supporting principles are:

- **Be consistent:** Don't make similar things different, including in spelling, behavior, or capability. Don't make different things appear similar when they have different behavior or capability. — For example, this proposal provides a single syntax for efficiently passing all "in" parameters, instead of teaching multiple divergent syntaxes for "in" parameters. It also avoids today's design problems of making `&&` reference parameters mean two incompatible things (rvalue vs. forwarding reference), making `&` reference parameters conflate two different meanings (`inout` and `out`), and having to explain that top-level `const` on a by-value parameter is meaningless except when it isn't. It also allows using "`= initializer`" consistently for construction and assignment, and guaranteeing that every variable is initialized before use with a program-meaningful value (never a dead write).
- **Be orthogonal:** Avoid arbitrary coupling. Let features be used freely in combination. — This proposal enables freely and conveniently composing `move` and `forward` with generic and concrete types equally, including easily declaring an "rvalue reference" parameter of a generic type and a "forwarding reference" parameter of a concrete type, both of which are difficult to express today. It makes `in`, `inout`, `out`, `move`, and `forward` independent and have distinct argument requirements, so any combination can unambiguously overload. It allows = to be used with the same meaning orthogonally to set an object's value, whether the value is being set at declaration time or not, and whether it performs construction or assignment.
- **Be general:** Don't restrict what is inherent. Don't arbitrarily restrict a complete set of uses. Avoid special cases and partial features. — This proposal supports extending the parameter passing model generally throughout the language where an object is passed from one piece of code to another, including range-`for` loops. It also supports for the assignment operator all the options already allowed for constructors, including assignment from zero and multiple arguments, to remove C++'s current arbitrary restriction.

These also help satisfy the principles of least surprise and of including only what is essential, and result in features that are additive and so directly minimize concept count (and therefore also redundancy and clutter).

Additional design principles include: Make important things and differences visible. Make unimportant things and differences less visible. — This proposal makes `inout` and `out` distinct, and suggests considering making arguments taken by mutable reference more visible at call sites.

0.4 Acknowledgments

Thank you to the following for reviews and comments on this material: Joe Bialek, Marshall Clow, Gabriel Dos Reis, Timur Doumler, Joel Filho, Lee Howes, Tom Hulton-Harrop, Alexander Jones, Nicolai Josuttis, Piotr Artur Klos, Thomas Köppe, Alisdair Meredith, Gor Nishanov, Barry Revzin, Nathan Sidwell, Michael Spertus, Bjarne Stroustrup, Andrew Sutton, Shane Tapp, Daveed Vandevoorde, Tim Van Holder, Titus Winters, and several reviewers whose names I don't know who commented via GitHub issues.

1 "What, not how" parameter passing

1.1 "What": Directly declare intent

> *"References were introduced primarily to support operator overloading. … C passes every function argument by value, and where passing an object by value would be inefficient or inappropriate the user can pass a pointer. This strategy doesn't work where operator overloading is used. In that case, notational convenience is essential…" — [Stroustrup94] p. 86*

This paper proposes to upgrade our "how" convention to an abstraction: By enabling the programmer to directly declare their intent of "what" they want to use the parameter for, we can leave the optimal delivery mechanics to the language and machine target, improving performance and correctness as well as simplicity.

This paper proposes five explicit ways to pass parameters, roughly in order from most to least commonly used:

```
auto f(     in X x) { }       // present me an X I can read from
auto f(  inout X x) { }       // present me an X I can read and write
auto f(    out X x) { }       // present me an X I will assign to

auto f(   move X x) { }       // present me an X I will move from
auto f(forward X x) { }       // present me an X I will pass along
```

Note Syntax is bikesheddable. This syntax is unambiguous when parameter names are required.

And it proposes two ways to return values:

```
auto f () move    X { }       // move an X to the caller
auto f ()         X { }       // (possibly same; we're already partway to 'move' default)

auto f () forward X { }       // pass along an X to the caller
```

The need for declaring the data flow direction has been rediscovered many times, in different languages (e.g., Ada and C#) and in C and C++ commercial projects using ad-hoc convention or nonportable annotations for tools like Doxygen and SAL. As UX study participants were quick to point out, when they are annotations they always get out of sync, whereas when they are in the language they stay in sync and enable compiler optimizations.

1.1.1 Implementation: Delegation to platform ABI (see also Appendix A)

The platform ABI specifies certain details of the argument passing convention. This paper describes the semantics, and the ABI specifies how those semantics are implemented. In some cases, the ABI has the option of specifying that the implementation be via an overload, or by passing additional bits in available CPU flags or combined in an additional register.

For example, for an `in X` or `move X` parameter: If `X` is cheaply trivially copyable (and therefore nonpolymorphic; e.g., `int`) it is passed by value, the platform ABI defines what is the limit of "cheap" (e.g., size is 64 bits or smaller, possibly other characteristics). Otherwise, `X` is passed by pointer with the semantics that rvalue arguments are automatically moved from, and for `in X` the platform ABI defines whether that is accomplished by emitting overloads on `const&` and `&&` (as we teach people to today by hand, but which is combinatorial) or by passing an `is_rvalue` flag for the parameter that is filled by the caller (and if so, where the flag is stored, in a CPU flag or combined in an additional register with other parameters' flags if there are any). If flags are passed in registers, the ABI documents how bits related to different parameters are combined in a bitfield to minimize stack frame overhead.

1.1.2 Summary of parameter passing semantics

The following table summarizes the semantics of each of the five declared intents. Overloading is allowed, and the compiler will select the best match based on the argument restrictions. The following sections go into more detail for each including examples.

Note that all of the following are compatible with calling existing functions. For example, an `in` parameter is simply treated as a `const` lvalue for all purposes including for passing it onward to an existing overload set, except that the definite last use treats it as an rvalue (only if the actual argument is an rvalue and the use would be legal for an rvalue) which naturally selects any rvalue overloads for the function it's being passed to.

For the last two, treating the parameter as a `const` lvalue for non-last uses allows the function to read the 'in' value (e.g., `forward` parameters commonly want to do trace/log output of the about-to-be forwarded value).

Highlights here and in §1.3.1 and §1.3.4 are speculative additions for possible extension to destructive move.

`____ X x`	**Calling convention**	**Caller arguments**	**Callee parameter uses**
`in`	`X` (by value) if cheaply trivially copyable (defined by platform ABI) Otherwise, `X&`	Any initialized object (lvalue or rvalue)	`x` is treated as a `const` lvalue, except that if the argument is a non-`const` rvalue then each definite last use of `x` that would be legal for an rvalue treats `x` as a non-`const` rvalue (as-if casting it to an xvalue) (including can move from an rvalue argument)
`inout`	`X&`	Any initialized non-`const` lvalue	`x` is treated as a non-`const` lvalue If the function is nonvirtual, at least one path must contain a non-`const` use of `x` (otherwise, the parameter should be `in`)
`out`	`X&`	Any non-`const` lvalue	`x` is treated as a non-`const` lvalue Every path must contain a definite first use of `x`, and that definite first use must be either to pass `x` to another function's `out` parameter, or else be an assignment-expression to `x` (otherwise, the parameter should be `inout`) If the first use is an assignment-expression, then if the argument is uninitialized it constructs `x` with the right-hand side expression treated as the constructor argument list; otherwise, it performs an assignment as usual
`move`	`X` (by value) if cheaply trivially copyable (defined by platform ABI) Otherwise, `X&`	An initialized non-`const` rvalue or initialized definite last use lvalue	`x` is treated as a non-`const` lvalue, except that each definite last use of `x` treats `x` as a non-`const` rvalue (as-if casting it to an xvalue)
`forward`	`X&`	Any object (lvalue or rvalue)	`x` is treated as a `const` lvalue, except that each definite last use of `x` preserves the argument's cv-qualification and value category

Notes No call site ever performs a non-trivial copy or move for an argument to these five parameter kinds.

Currently above, the "pass small trivially copyable types by value" optimization is only applied to `in` and `move`. We are considering generalizing it to perform this optimization uniformly in both directions which would additionally cover `inout`… the generalization for small trivially copyable types is: (a) pass all "in" by value (includes not only `in`, but also the "in" part of `inout`, and `move`) and return all "out" by value (includes `out` and the "out" part of `inout`). With this generalization, an `inout` parameter that is small and trivially copyable would not have to be passed by an indirection at all as it must today, but could be passed by copy in and by copy out which can be more efficient because then the `inout` parameter could be passed entirely in registers.

Polymorphic types are naturally passed by reference in all cases. For example, `in base` is perfectly sensible and equivalent to `base const&`. In some cases this proposal adds new expressive power; for example, `forward base` (or any specific non-template-parameter type) is hard to express today, but here it works fine to express a desire to forward an object whose type any of a subset of related types, and forwards the argument object's const-ness (which is useful) and value category (which is likely not useful for a noncopyable type, but doesn't hurt).

1.2 Definite first/last use (see also P1179)

This paper uses the following concepts, which overlaps with the [P1179R1] Lifetime rules. Other modern languages already successfully use similar path-based rules, such as C#'s definite assignment rules.

A "**local variable**" means any parameter, local variable, or temporary object in the scope of a function definition.

A function's "**control flow graph (CFG)**" has its usual meaning, except herein we ignore exceptional paths. A "**node**" has its usual meaning in a CFG except that each subexpression that mentions a local or nonlocal variable (including a local variable's declaration or destruction) is treated as a distinct node (rather than combined in larger basic blocks). A "**path**" is a connected directed sequenced-before path of nodes in a function's CFG.

"**Use**" of a local variable `var` means a node corresponding to a mention of `var`'s name that accesses `var`'s value (not its declaration, not just taking its address). A "**`const` use**" means a use that only reads `var`. A "**non-`const` use**" means a use that is not a `const` use (such as passing `var` to a function that takes it via reference to non-`const`). A "**definite first (or, last) use**" of `var` means a use that is not preceded (or, followed) by any other use of `var` on any path that shares that node, and is not in a loop body or on any path that traverses a backward `goto`.

When a local variable `x` has accessible data members `x.y` and `x.z`, then a definite first (or, last) use of `x.y` means a use that is not preceded (or, followed) by any other use of `x.y` or of `x` as a whole (but not affected by uses of `x.z`).

1.2.1 Example

For example:

```
void f(int i, int j) {
    use(i);                     // definite first use of i
    if (cond) {
        if (cond2)
            use(i,j);           // definite first use of j
    } else {
        use(i,j);               // definite first use of j
    }
    use(i);
    use(j);
    for (auto x : {1,2,3}) {
        if (cond3)
            use(x,i,j);
    }

    if (cond3)
        use(j);                 // definite last use of j
}
```

Notes A local variable can have zero or more definite first uses, and zero or more definite last uses.

A variable may not have a definite first or last use on every path. In the above example, `i` does not have a definite last use on any path. Some kinds of parameter passing described in this document require that the parameter does have a definite first and/or last use on every path.

These rules will be simple for programmers to reason about, because they correspond directly to a simple linear visual scan of the code's nested blocks. C# programmers, who greatly value simplicity, have found C#'s "definite assignment" easy to reason about, and that feature's 12-page specification (see [ECMA-334], section 10.4.4, pp. 77-88) is significantly more complex than the above.

1.3 Parameters

1.3.1 `in` parameters

*"An internal memo dated January 1981 [Stroustrup,1981b] describes the idea: 'Until now it has not been possible in C to specify that a data item should be read only, that is, that its value must remain unchanged. Neither has there been any way of restricting the use of arguments passed to a function. Dennis Ritchie pointed out that if **readonly** was a type operator, both facilities could be obtained easily…' " —B. Stroustrup (D&E, p. 89)*

```
auto f(in X x) { }            // present me an X I can read from
```

An `in` parameter is a parameter that the function body can read from. This case optimizes based on the size of the object and the lvalueness of the argument. For example:

C++20 equivalent	This paper (proposed)

`void f1(const X& x) {`
`  g(x);`
`}`
`void f1(X&& x) {  // overload to optimize rvalues`
`  g(std::move(x));  // remember to move only once`
`}`
	`void f1(in X x) {`
	`  g(x);`
	`}`

`template<typename T>`
`void f2(const T& t) {  // can't easily overload for rvalues, &`
`  g(t);  // nobody writes the requires/enable_if`
`}  // to pass builtins by value...`
	`template<typename T>`
	`void f2(in T t) {`
	`  g(t);`
	`}`

`template<typename T> bool should_pass_by_value_v`
`  = std::is_trivially_copyable_v<T> && sizeof(T) < _SOME_MAX;`
`template<typename T> requires should_pass_by_value_v<T>`
`void f3(T t) {`
`  my_trace("f", t);`
`  container.emplace_back(t);`
`}`
`template<typename T> requires (!should_pass_by_value_v<T>)`
`void f3(const T& t) {`
`  my_trace("f", t);`
`  container.emplace_back(t);`
`}`
`template<typename T> requires (!should_pass_by_value_v<T> &&`
`  !std::is_lvalue_reference_v<T>)`
`void f3(T&& t) {`
`  my_trace("f", t);  // teaching: why no move here?`
`  container.emplace_back(std::forward<T>(t)); // erm, "move"`
`}`
	`template<typename T>`
	`void f3(in T t) {`
	`  my_trace("f", t);`
	`  container.emplace_back(t);`
	`}`

Highlights here and in §1.1.2 and §1.3.4 are speculative additions for possible extension to destructive move.

For `in`, as also described for `out` (see §1.3.3), whether or not an exception is thrown, the function informs the caller whether an rvalue argument was moved from using a `[[relocates]]` move, and if so and the caller argument was a definite last use then the caller can elide its destructor.

Notes This lets us improve current C++ parameter passing guidance for "in" parameters in several ways.

First, it makes writing optimal code simple (and the default) for passing cheap-to-copy objects. In C++ we have always taught people to pass "small" or cheap-to-copy objects by value as `(X x)`, and "large" expensive-to-copy objects by reference as `(const X& x)`, and then teach what "small" means which changes over time on different platforms. Today's guidance on this is not consistent; some authors teach "pass only built-ins by value," others teach "pass built-ins or small-looking objects by value," and both may be dated because actual measurements on modern platforms show that larger-than-historical objects are cheaper to pass by value which is another reason to leave it to the platform ABI. In this design, the programmer just passes `(in X x)` and the platform ABI specifies what "cheap to copy" means in a deterministic way for that platform ABI. Not only do programmers get to write simpler code and no longer have to remember this overloading-style guidance (at all and especially not per-platform), but they also get better portable performance as the same code is automatically implemented appropriately on each given platform.

Second, this is especially useful for templates. Essentially nobody tries to apply `requires` or `enable_if` and overloading to optimize cheap-to-copy `T`. – In this proposal, `in` does that naturally.

Third, it makes the "in+copy" case easier where the function will (or may) keep a copy of the parameter. In C++20, for this case we teach programmers who need to optimize for rvalues to overload on `(const X& x)` and copying from `x` in the body, and `(X&& x)` and writing `std::move(x)` in the body and on the last use only, with the function bodies usually otherwise identical. The C++20 standard library contains many examples of this duplication. – In this proposal, no hand-written overload is required, and in the body of the function we do not have to remember whether to use an explicit `std::move`, and call sites can correctly elide destructors after last use for moved-from objects when a relocating move is invoked (see §1.3.3).

Fourth, it avoids the long-standing wart in C and C++ that `f(T t)` and `f(const T t)` declare the same function. The confusion does not arise here because "in" parameters are never `const`-qualified and are always treated `const` in the function body.

Finally, it enables an additional optimization in environments that perform calls outside the program's address space and so always copy data to/from the callee (e.g., RPC, sending data to GPU memory), because `in` lets the compiler automatically elide copying in one direction by construction.

What about a function that currently passes `T*` to express an optional `T` parameter? Leaving it as `in T*` is fine, or `in optional<T>`; two other options are overloading and default arguments, as today. Pointers are still a fine thing to pass when you want to do that to say "what" is being passed, in this case an optional parameter. The main place this proposal would remove raw `T*` parameters is when they are being used to express "how" not "what," such as for `out T` parameters.

There is one design point that we should validate with field experience: Are there reasons to guarantee a copy is performed?

1) If an "in" parameter is aliased by another parameter or global, then although the `in` parameter is treated as `const` in that it cannot be modified directly, its value may change through the alias. This could be a pitfall, which we should determine by getting field usability experience. However, the [P1179R1] Lifetime rules default to banning passing non-owning Pointers that alias, so this may not be a problem in practice.

2) For coroutines, an `in` parameter passed by pointer (much like today a `const&` parameter) can be invalidated/dangling at the first suspension point, particularly if it binds to an rvalue argument. This case is intended to be covered by [P1179R1] Lifetime rules extended for coroutines, where after the first suspension point we would warn about any use of a `const&` or `in` parameter (and potentially other by-pointer passing).

3) What if we learn some other reason why programmers want to be able to guarantee a copy? If the function body is going to copy to somewhere else anyway, this is mostly moot since the copy will occur then. So I think the real question here is whether a local use of the parameter within the function is guaranteed to have a separate identity from the argument, such as if you care about aliasing (above). In the current proposal without a `copy` option, you can get the nearly identical effect now by passing as `in` and then just copying into a local variable and using that. That "nearly" has three parts:

a) If `in` is going to copy already, this results in two copies. However, both will be trivial/cheap copies, and appear to be regularly optimized away at -O1, as in this Godbolt example.

b) If `in` is going to pass by reference already, this results in exactly one copy either way. However, passing by `in` and then copying adds a pointer to the space/time overhead of calling the function, and this appears to be regularly optimized away at -O1 by GCC but not optimized at all by Clang even at -O3, as in this Godbolt example.

c) It's an idiom, which doesn't directly express intent. If this turns out to be relatively common, that would be an argument for language support.

We can consider treating a local variable as an rvalue on definite last use, similar to an `in` parameter. Then examples like this would move:

```
void f(in X);

void g() {
    X x;
    f(x);           // definite last use of x, so treat as rvalue
}

void f(in X x) {
    X x2 = x;       // this is a move if the argument is an rvalue
}
```

1.3.2 `inout` parameters

```
auto f(inout X x) { }          // present me an X I can read and write
```

An `inout` parameter is a parameter that the function body can read and write. At least one path must contain a non-`const` use of `x` (else the parameter should be `in`). For example:

C++20 equivalent	This paper (proposed)

`void f1(/*inout*/ X& x) { // can't distinguish inout v out`
`  g(x);  // ok`
`  ++x;  // ok modifies but can omit`
`}`
	`void f1(inout X x) {`
	`  use(x);  // ok`
	`  ++x;  // ok modifies and required`
	`}`

`void f2(/*inout*/ X& x) { // can't distinguish inout v out`
`  y = x * 2;  // ok`
`} // not flagged: did not write to x`
	`void f2(inout X x) {`
	`  y = x * 2;  // ok`
	`} // error, did not write to x`

Note This is clearer than current C++ guidance. Today, both "in-out" and "out" are expressed by passing `(X& x)`, so they are neither distinguished to the reader of the code nor enforceable by the language. Yet that is an important distinction to make. In this design, the programmer explicitly distinguishes "in-out" which allows reading from the parameter, and "out" which enforces that the parameter is not read until it is first written to.

1.3.3 `out` parameters

*"Finally, the [internal January 1981] memo introduces **writeonly**: 'There is the type operator **writeonly**, which is used like **readonly**, but prevents reading rather than writing…' "*
—B. Stroustrup (D&E, p. 90)

```
auto f(out X x) { }          // present me an X I will assign to
```

An `out` parameter is a parameter that the function body can read from only after writing to it, and will assign to it. (Note: The argument may be **uninitialized**; see §2.1.) For example, assuming `g(a)` has a non-`out` parameter:

C++20 equivalent	This paper (proposed)

`void f1(/*out*/ X& x) {  // cannot distinguish inout vs. out`
`  g(x);  // not flagged: read`
`  x = 42;  // ok but can omit`
`  g(x);  // ok`
`}`
	`void f1(out X x) {`
	`  g(x);  // error, trying to read "in" value`
	`  x = 42;  // ok, required`
	`  g(x);  // ok`
	`}`

`void f2(/*out*/ X& x) {  // cannot distinguish inout vs. out`
`  // ... no write to x ...`
`} // not flagged: did not write to x`
	`void f2(out X x) {`
	`  // ... no write to x ...`
	`} // error, failed to write to x`

If no exception is thrown, the function is guaranteed to have constructed an uninitialized argument. If an exception is thrown, if the argument was uninitialized it remains uninitialized (if it was initialized before the exception was thrown it is destroyed again in the callee).

For a parameter `out X x`, if the assignment-expression `x = expr;` is not a valid call to `operator=`, rewrite the call to `x = X{expr};` as a fallback.

Note The canonical performance motivation for `out` is to reuse the capacity of a container like `string` or `vector` when calling `read_next_chunk` in a loop (see also an overlapping Note in §2.1):

```
vector<thing> v;
while (more) {
    read_next_chunk(v);
    process(v);
}
```

With an `out` parameter (today, a `&` parameter) we only need to reallocate if the capacity isn't already big enough. So typically for this entire loop will only allocate memory on a few calls to `read_next_chunk`, and then there are no allocations at all once we get to the biggest chunk for that loop. That's a big performance savings compared to returning by value which forces an allocation on every call. In the degenerate case where all the chunks are the same size, we allocate only once for the whole loop.

Inside `read_next_chunk`, the loop body may already have a suitable string or character buffer it can just assign from. For example, `string::operator=` can assign from anything we can view with a `string_view`, and so it can copy from the contents of any contiguous character buffer without any allocation or temporary `string` object if the capacity is already big enough.

In cases where the type doesn't have a suitable `operator=`, you can usually use the technique of `x = {};` (default assignment, see §2.3; this is usually very cheap and for types like `string` and `vector` won't change the capacity) followed by using any non-`const` functions to set specific values in `x`.

Both ways, we still have the nice guarantee on exit that all elements in the `string` or `vector` really are ones that the function wrote to, avoiding the usual problems with fill-array APIs regarding 'but how many elements did the function actually write to and how many did it not which might be uninitialized' – this is not a problem with this technique.

1.3.4 `move` parameters

Highlights here and in §1.1.2 and §1.3.1 are speculative additions for possible extension to destructive move.

```
auto f(move X x) { }          // present me an X I will move from
```

A `move` parameter is a parameter that the function body will move from (consume the value of). For example:

C++20 equivalent	This paper (proposed)

`void f1(X&& x) {`
`  container.emplace_back(std::move(x));`
`}`
	`void f1(move X x) {`
	`  container.emplace_back(x);`
	`}`

`template<typename T> // rref not allowed for templated T`
`void f2(T&& t) {`
`  container.emplace_back(std::forward<T>(t));`
`}  // so use forward instead of move`
`// + overload and =delete (const&) and (&) for lvalues`
	`template<typename T>`
	`void f2 (move T t) {`
	`  container.emplace_back(t);`
	`}`

If no exception is thrown, the function is guaranteed to have moved from its argument. Whether or not an exception is thrown, the function informs the caller whether the argument was moved from using a `[[relocates]]` move, and if so and the caller argument was a definite last use then the caller can elide its destructor.

Notes This last paragraph is symmetric with `out`'s guarantees.

Many existing movable types, including `std::` smart pointers, already leave them in a non-owning state that does not require a destructor call (see P1029 for analysis). To be certain, however, we can require adorning a move operation with explicit `[[relocates]]` to designate a relocating move, which then triggers this optimization to automatically not destroy after a moved-from last use.

I'm currently naming this `move` for familiarity (and, unlike today, actually does guarantee a move), including in §0 regarding a potential call-site `move` qualifier instead of `std::move`. I think `consume` may be a better name, for the parameter and for any §0 call-site qualifier.

Declaring the parameter passing intent directly avoids the problems of expressing it indirectly today in terms of `const` and references, for example that we avoid today's problem that we have to explain why `const &&` rvalue references don't make sense, and `&&` rvalue references are onerous to write for templates.

This should eliminate the need to write `std::move` except when move will actually occur. As Bjarne Stroustrup notes, in today's teaching we recommend against the use of `std::move` just as we recommend against other casts; with this feature, we can do that more consistently. And when we do write `move`, this lets us teach "yes, `move` means `move`." (See also §0 regarding `move` at call sites.)

As Peter Dimov notes, we have a special rule for `return` (and, more recently, `throw`) that can be generalized so that the definite last use of an object is automatically made move-eligible. The problem is that we cannot easily do this in existing code without breakage; these new features enable us to do it without breakage because it applies to code that does not exist today. Please see [lib-ext 2017-02-08] and the followup replies.

Also, we no longer have to explain that "requests to move fall back to copy" with special rules. Instead, it's just overload resolution on `in` and `move`: `in` can take any argument, and `move` takes only non-const rvalues, so if we have a call site that is `move` request and there is no `move` constructor in the overload set we naturally just select the `in` (copy) constructor by simple overload resolution, with no need for a special fallback rule.

1.3.5 `forward` parameters

```
auto f(forward X x) { }       // present me an X I will pass along
```

A `forward` parameter is a parameter that the function body will pass along to other code, and preserves the argument's cv-qualification and value category. The function may forward a value of type `X` to other code (once), but not use the variable itself in any other way that could disturb it. The function's body may read from `x` before it forwards it. For example:

C++20 equivalent	**This paper (proposed)**

`template<typename T>`
`void f1(T&& t) {`
`  container.emplace_back(std::forward<T>(t));`
`}`
	`template<typename T>`
	`void f1(forward T t) {`
	`  container.emplace_back(t);`
	`}`

`// difficult to write for non-templates`
`template<typename widget>`
`  requires is_same_v<remove_cvref_t<T>, widget>`
`void f2(widget&& w) {  // move a concrete type 'widget'`
`  container.emplace_back(std::forward<widget>(w));`
`}`
	`void f2(forward widget w) {`
	`  container.emplace_back(w);`
	`}`

Notes This allows a forwarding parameter without writing a template.

This also makes it easier to replace function-like macros. This macro:

```
#define FOO(x,y) ((x) + (y))
```

can be replaced with the following function template in C++17:

```
template<typename X, typename Y>
auto Foo(X&& x, Y&& y) -> decltype(auto)
    { return std::forward<X>(x) + std::forward<Y>(y); }
```

and can be replaced with the following equivalent in C++20:

```
auto Foo(auto&& x, auto&& y) -> decltype(auto)
    { return std::forward<decltype(x)>(x) + std::forward<decltype(y)>(y); }
```

and can be replaced with the following equivalent in this proposal:

```
auto Foo(forward auto x, forward auto y) -> forward { return x + y; }
```

1.3.6 (optional) Call site explicit side effects

Optionally, we could also consider requiring that the call site be annotated in some way for every argument to an `inout`, `out`, `move`, or `forward` parameter, to avoid the common problem that the call site has no visual indication that the object it passes could be modified.

For example, given:

```
void f1(in      A a);
void f2(inout   B b);
void f3(out     C c);
void f4(move    D d);
void f5(forward E e);
```

We could default every argument to such a parameter to `const` unless explicitly annotated `out` (or possibly `mutable`) to opt into non-`const`:

```
int main() {
    A mya; B myb; C myc; D myd; E mye;

    f1(mya);           // ok

    f2(myb);           // error: myb is treated as const by default
    f2(out myb);       // ok (or "f2(mutable myb")) or "f2(&myb)" etc.)

    f3(myc);           // error: myc is treated as const by default
    f3(out myc);       // ok (or "f3(mutable myc") or "f3(&myc)" etc.)
```

For move parameters, we already require `std::move`, and could add `move` as a convenience syntax, to explicitly opt into rvalue-ness:

```
    f4(myd);           // error: myd is an lvalue
    f4(std::move(myd)); // ok (already required in C++20)
    f4(move myd);      // potential alternative
```

For `forward` parameters, which transparently forward onward to other code, this allows additional expressivity that's otherwise inconvenient to write:

```
    f5(mye);           // ok: treated as const, forwards only value category
                       //     (f5 can only forward its argument to const functions)
    f5(out mye);       // ok: forwards constness and value category
    f5(move mye);      // ok: forwards as non-const rvalue
}
```

This trades off some inconvenience for clearer and more robust code. I see strong arguments on both sides.

However, this can lead to a lot of verbosity if required for every non-const argument. For example, these examples noted by "stan423321" in this paper's Issue #31:

```
std::swap(out a, out b);
out std::cin >> out c >> out d;
out b += c;
out mymap .clear();
out mymap [a] = b;
(out (out mymap)[a]) += d;
```

Pro arguments:

- **Readability:** Makes it easy to see the flow of data without having to look at the function declaration.
- **Correctness:** Addresses major classes of persistent real-world errors including exploited security vulnerabilities.
- **Prior art:** Various C++ environments have reinvented these modifiers for `&` to non-`const` parameters using nonportable syntax or annotations. Various C++ coding standards encourage or require passing by `*` to non-`const` instead of `&` to non-`const`, even if nullness should not be allowed, in order to force call sites to visibly write `f(&arg)` instead of just `f(arg)` to make it visible that there's more going on than just reading `arg`. There is also positive field experience with similar features in other widely used languages (e.g., C# `ref` and `out`, which are required on both the caller and the callee).

Con arguments:

- **Noise:** This could be viewed as "needless verbosity" and "nannying the programmer." This concern should be weighed against the repeated reinvention and good experience in C++ and other languages in the "Pro arguments" above, and is possibly mitigated by that the most common parameter passing, `in`, does not require any annotation.

> **Note** If we added these call-site annotations, they should initially be mandatory. If experience shows we prefer making them optional (e.g., like `override`) then they can always be relaxed later without a breaking change, but the reverse would be a breaking change (e.g., with `override`, some have argued that it should ideally be required, but we cannot make it required because of backward source compatibility).

Relatedly, we could permit `forward` argument modifier at a call site to denote explicitly forwarding a `forward` parameter when the function wants to forward it multiple times. Consider this example from Joel Filho:

```
void invoke_twice(forward auto f, forward auto... args){
   std::invoke(f, forward args...); // annotation required for the code to work
   std::invoke(f, args...);         // definite last use, automatically forwards
}

int x = 0;
auto increment = [](auto& val){ ++val; };
invoke_twice(increment, x);
```

1.3.7 Argument traits

Several of the proposed declarative parameter passing conventions' semantics require additional information to be communicated from the caller to the callee, and these are available via trait queries:

`____ X x`	**Argument trait information, known statically at call sites and queryable dynamically in callees**
`in`	`bool arg_is_nonconst_rvalue(x)`
`inout`	—
`out`	`bool arg_is_uninitialized(x)`
`move`	—

`forward`	`bool arg_is_nonconst(x)`
	`bool arg_is_rvalue(x)`
	`bool arg_is_uninitialized(x)`

Note The directions of the boolean flag meanings for `in` and `out` are chosen so that they are `false` in the common case, so FFIs from other languages can more easily pass `false`/`0`-bits by default.

See Appendix A for implementation options. The current approach is to implement `forward` statically and `in` and `out` dynamically, which means the three `forward` traits can be `constexpr`. Additionally, when `in` is implemented using pass-by-value, the `is_nonconst_rvalue` flag is not needed (and should not be generated) and the trait can unconditionally return `false`.

1.4 Range-for

A range-for loop passes each object in the range as an argument to the loop body, and the loop iteration variable is already declared as a parameter. Applying the intentional parameter passing declares the loop's intent:

```
for (     in X x : rng) { }              // an X I can read from
for (  inout X x : rng) { }              // an X I can read and write
for (    out X x : rng) { }              // an X I will assign to
for (   move X x : views::move(rng)) { } // an X I will move from (move iters required)
for (forward X x : rng) { }              // an X I will pass along
```

This enables declaratively self-documenting loops that directly declare things we cannot conveniently express in C++20, and so achieve better correctness and/or efficiency:

- Loops that read elements are correct and efficient, and convenient to write: A C++20 `for(auto x : rng)` loop copies each element which is a known performance pitfall for large objects, and allows modifications to `x` that will be silently thrown away because programmers rarely write `const` here to express a read-only loop. Alternatively, a C++20 `for(const auto& x : rng)` loop passes every element by reference even when copying would be more efficient. – In this proposal, `for(in auto x : rng)` automatically is both efficient and guaranteed to be read-only.
- Loops that modify elements can be stated explicitly and conveniently up front using `inout`.
- Loops that `move` from the elements in the range can be stated explicitly up front, and require `move_iterator`s. In the foregoing, `views::move` is the range-v3 library's name for a convenience wrapper that creates a pair of `move_iterator`s from a range, the same way that `std::make_move_iterator` wraps an individual iterator.
- Loops that `forward` the elements in the range can be nongeneric: A C++20 `for(auto&& x : rng)` must be generic, whereas this design permits both the generic `for(forward x := rng)` and the nongeneric `for(forward x : X = rng)`.
- Loops can correctly initialize an array or other collection: A `for(out x := rng)` loop is guaranteed by construction to visit every element of `rng` and set it (construct or assign as appropriate) without reading the initial value (if any), and can deterministically initialize objects in an array.

1.5 Return values

These are primarily proposed for symmetry and to make current spellings more consistent, for example to spell the unique `->decltype(auto)` as a more regular `->forward`.

1.5.1 `move` return value

Given

```
auto f() -> move X { }        // move an X to the caller
```

we have a function that returns an rvalue of type `X`.

> **Note** The default value return is already on a path to these semantics, including move from `return` of a local variable, and later move from `throw` of a local variable. If we like it, we could go further in the same directly by making this the whitespace default.

1.5.2 `forward` return value

A `forward` return value preserves the return-expression's cv-qualification and value category. A function definition is required.

Given

```
auto f() -> forward X { }     // pass along an X to the caller
```

we have a function that forwards a value of type `X`, preserving its cv-qualification and ref-qualification as deduced from the `return` statement. Explicit cv-qualification or ref-qualification is not allowed.

> **Note** When returning an lvalue, this preserves the ability to naturally use lvalue returns in compound expressions (e.g., `v[0] = 42;`) including function chaining (e.g., `f(a).g(b).h(c);`).

A deduced return type defaults to `forward X` if all returns in the body return the same `forward` parameter of (possibly deduced) type `X`.

1.6 Overloading

Here is the matrix for viable arguments for each parameter type, according to whether the argument's constness, l/r-valueness, and initialization state (initialization applies to local lvalues only).

	uninitialized	**const lvalue**	**non-const lvalue**	**const rvalue**	**non-const rvalue**
in	-	ok	ok	ok	ok
inout	-	-	ok	-	-
out	ok	-	ok	-	-
move	-	-	-	-	ok
forward	ok	ok	ok	ok	ok

Overloading on only these parameter passing options is not allowed. For example:

- Overloading `f(in X)` and `f(out Y)` is allowed, because they have different types.
- Overloading `f(int X)` and `f(out X, in Y)` is allowed, because they have different numbers of parameters.
- Overloading `f(in X)` and `f(out X)` is not allowed, because they differ only in `in` vs. `out`.

Mixing these parameter passing options with today's pass-by-value/`&`/`&&` is not allowed in an overload set of functions with the same name in the same scope.

Notes Overloading `in` and `out` is naturally ambiguous because those two do not accept a subset or disjoint set of arguments, and so overloading cannot be allowed. (If we don't adopt and require §0.)

Otherwise, all combinations have unambiguously better matches for any call site and so in principle "could" be allowed to overload. Similarly, mixing uses of these and the pass-by-value/`&`/`&&` styles "could" be allowed in an overload set because they are either naturally ambiguous or naturally unambiguous based on the constness and lvalue-ness of the call site's arguments. But just because we can doesn't mean that we should, because the goal is simplicity and to expose combinatorial expressiveness only when it is actually useful — so, "should" overloading these be allowed?

These parameter passing options now directly express the programmer's intent of what the parameter is used for, including especially the direction of data flow and whether/what side effects are allowed on the argument. An overload set should have consistent semantics so that a call site can reason about *the whole overload set's meaning*, including things like whether the argument can be modified or whether its ownership is transferred. So it makes much less sense to overload on only the parameter passing option, because if `f(t)` (including in a template) could have totally different effects depending on the value category of `t` then the call site can't reason well about the meaning of the call.

Today, the most compelling and known-useful example of overloading these is to overload a function on `const&` ("in") and rvalue-`&&` ("move"). However, the only reason we do that today is for the purpose of optimizing for rvalue arguments… which `in` now already does, which removes the known motivation for overloading `in` and `move`.

If we discover compelling use cases to overload these, we can easily relax this restriction in the future and allow the overloading, except only the only inherently ambiguous combination of `in` and `forward`.

1.7 Segue: The `this` parameter

For member functions, the `this` parameter is implicit and so can't be qualified using the same syntax as other parameters, and so the C++ convention is to express qualifiers on the `this` parameter via special additional grammar at the end of the member function parameter list. Declarative parameter passing for `this` would then go in the same place as all the other `this` qualifiers:

```
// qualifying "this" on member functions
auto f() in      { }          // present me an X I can read from
auto f() inout   { }          // present me an X I can read from and will write to
auto f() out     { }          // present me an X I will assign to
auto f() move    { }          // present me an X I will move from
auto f() forward { }          // present me an X I will pass along
```

Note how these correspond to, and subsume and extend, the existing member function qualifications:

Passing `this` object…	**Similar to today's…**	**Notes**
`auto X::f() in`	`auto X::f() const`	`in` additionally passes `*this` by value when that is cheaper
`auto X::f() inout`	`auto X::f()`	Today's existing "mutable" default
`auto X::f() out`	`X::X` (constructor)	See §2
`auto X::f() move`	`auto X::f() &&`	`move` additionally guarantees that if the function returns normally, `this` object is known to be moved-from
`auto X::f() forward`	n/a	Today there is no way to forward `this` object

2 Generalizations: Guaranteed and unified initialization

2.1 `out` enables consistent `= uninitialized`

As a related feature, a variable of any type may be explicitly uninitialized by declaring it `= uninitialized` with the simple semantics of deferring the constructor call to the definite first use, which directly helps examples where a sensible program-meaningful initial value is not known at the point the variable must be declared:

- A declaration of a local variable `x` that is initialized with `= uninitialized;` allocates stack memory as usual but does not invoke a constructor.
- Every path containing `x`'s declaration must contain a definite first use of `x` before the declaration or initialization of another local variable, and that definite first use must be either to pass `x` to another function's `out` parameter, or else be an assignment-expression to `x` (which can be from a single-object expression or a possibly-empty `initializer_list`) that performs a single constructor call (usually direct-initialization) of `x` with the right-hand side expression treated as the argument to `x`'s constructor.
- If two local variables `x` and `y` are declared in that order and are explicitly uninitialized, then on every path `x`'s definite first use must precede `y`'s definite first use.

Notes For security applications, the initialization can also set the padding bits.

We always initialize in declaration order so that variable lifetimes continue to nest correctly. A later-declared local variable `y` can safely refer to a previously-declared variable `x` that will outlive it.

The following have exactly the same effect in terms of constructor/copy/move calls including copy/move elision, but the first offers more expressive flexibility:

```
X x = uninitialized; /*...*/ x = f();

X x = f();
```

For example:

```
int i;                          // i is uninitialized (current language rule)
int j = uninitialized;          // same, except guarantees j will be initialized
if (cond) {                     // allows true alternate initialization
    i = 1;                          // i is initialized
    j = 2;                          // j is initialized
} else {
    i = 100;                        // i is initialized
    j = 200;                        // j is initialized
}

std::string s = uninitialized;  // s is uninitialized, no constructor is called
if (cond2) {
    s = "xyzzy";                // s is constructed, calls std::string(const char*)
} else {
    // calls: fill_string(int val, out string s)
    fill_string(i+j, s);        // s is constructed, function will construct string
}
```

This guarantees initialization without having to eliminate dead writes for overwritten initial values. For example:

```
void fill_struct(out BigStruct s) {         // guaranteed initialization
    // computation, then the definite first use of s is:
    s = computed_value;                     // calls s's constructor
    // any const or non-const use of s is fine after this point
}

int f() {
    BigStruct data = uninitialized;         // no dead writes to eliminate
    do {
        fill_struct(data);                  // guaranteed to construct on first call, then
                                            // guaranteed to assign on each subsequent call
    } while (something);
}
```

Notes The output parameters of functions like `std::memset` and `std::copy` and similar functions should be treated as if declared with `out` parameters.

We still teach "declare variables as locally as possible" and "prefer to initialize variables at their point of declaration" most of the time; this proposal just also acknowledges the reality of situations where that advice is not appropriate, such as the examples later in this Note. Importantly, initialization of variables at their point of declaration continues to be implicitly encouraged with a major carrot: The language gives automatic type deduction support for the initialize-at-declaration case, whereas using `=uninitialized` requires naming a concrete (non-deduced) type.

Allowing an assignment-expression to invoke a constructor, and allowing an `out` parameter to call a constructor, is intentionally a step toward actual uniform initialization in C++ where all value-setting (construction and assignment) is spelled using "`= val`" syntax in all cases.

This lets us provide initialize-before-use guarantees, and also improve parameter passing guidance. First, it improves the correctness and efficiency of initializing large PODs, including arrays of PODs. Even though many coding styles encourage initialization-at-declaration, they universally make an exception for cases like (C++20 style)

```
db_format data;                 // POD matching a data layout structure
read_next_chunk(&data);         // e.g., from disk or network
while (...) read_next_chunk(&data);
```

(see also an overlapping Note in §1.3.3) where requiring a redundant artificial initial value on the declaration is **unuseful** (because the value will be immediately overwritten), **expensive** (because compilers cannot always eliminate the dead write), and **error-prone** (because the language provides no way to prevent `read_next_chunk` from reading from uninitialized data, and to require that it does write to at least some part of the data). – In this proposal, we allow functions like `read_next_chunk` to express that they guarantee to initialize (write to) the parameter, guarantee that they will not read from it, and enable strong initialization guarantees without requiring artificial initial values.

Changing the `read_next_chunk` to return by value instead is also not cost-free as it requires at least a move, and is generally a bad design when it is called repeatedly because the alternative to using a caller-provided buffer is to perform an allocation on every call or to resort to brittle workarounds like a static buffer.

Note that the "out" pattern is not limited to POD buffers, but comes up regularly in well-written code like (C++20 style)

```
vector<byte> data;              // nicely encapsulated non-POD type
read_next_chunk(data);
while (...) read_next_chunk(data);
```

where it is key that the function reuse a caller's storage instead of managing its own; this is what takes this kind of loop from (possibly large) *N* allocations down to a single allocation for the entire loop. The implementation of `read_next_chunk` does not need to perform any `const` operation on `data`; it will either assign to `data`, or `.clear` then `.push_back`, or `.resize(n)` then write through `[]` subscripting, or otherwise fill `data` using any other combination of non-`const` operations. Under the proposed design, this code and all of those techniques still work, but we can gain the guarantee that `read_next_chunk` does not read stale `data` and will actually store new `data` (important to avoid reading garbage data, or sensitive data).

Further, these are semantics you want for a potentially moved-from argument: If the caller passes a moved-from object, `out` expresses nearly all of the safe uses of that object, and no unsafe ones.

If there is motivation to distinguish initialized vs. uninitialized arguments, we could permit overloading on `out` and `inout`.

For trivially copyable types, there is an additional optimization in environments that perform calls outside the program's address space and so always copy data to/from the (e.g., RPC, sending data to GPU memory), because `out` lets the compiler automatically elide the first copy (to the callee's memory) because by construction no data is to be copied in.

Use `inout` to express "maybe-`out`" parameters that are not read but are written to, such an `error_code` parameter that is only set on error paths. The only additional thing that a `maybe_out` parameter style could provide would be to statically enforce that the callee doesn't read the input value (we could require that a definite first use must be at least a non-`const` operation, or even more strictly an assignment). At this point that one difference from `inout` doesn't seem to warrant adding a separate option, so I'm considering it as just one of the uses of `inout`; but we can always add it as a separate option anytime if we decide that it's important.

If the function throws an exception, it informs the caller whether an uninitialized `out` parameter was initialized.

If `X` is an array type, initializing every element of the array is required. However, the cost of doing so is greatly mitigated compared to today's C++ because we never initialize dummy values that will only be overwritten with "real" program-meaningful values.

In summary, `=uninitialized` together with `out` parameters gives us consistent initialization guarantees for PODs and non-PODs. Declaring `=uninitialized` has the same meaning as the default initialization for stack-based POD variables, but additionally:

- works consistently for **all types** (not just PODs);
- **statically guarantees initialization** on definite first use, including via an initializing function call via `out` parameter (a common, even pervasive, style of initialization with program-meaningful values);
- **enables initialization to always use program-meaningful values**, never requiring a placeholder (including default-constructed) value that is just injecting dead writes intended to be overwritten;

- guarantees that the later initialization is **consistently spelled as `=initializer`**;
- guarantees that the later initialization is an efficient **single constructor call**; and
- last but not least, lets us give **consistent guidance** for all types (instead of routinely giving different guidance for large PODs and arrays, which today we routinely teach not to initialize on declaration) and starts to put us on a path to a consistent spelling of initialization.

2.2 `out` is a named and unified { constructor, value-setter }

Because any function can initialize its `out` argument, any function with an `out` is effectively a named constructor. When the function is a non-member non-friend, it is always a 'delegating' constructor, that is, one that's written in terms of other constructors. When `this` is `out`, as in the following example, the named constructor has the usual private access.

Being able to conveniently give meaningful names to constructors naturally makes them both

- "explicit," because they are always a named function call; and
- "tagged," because they can be given distinct names;

and so subsumes those two usual workarounds for implicit or ambiguous conversions with a direct way to express the intent that also makes the resulting calling code clearer and self-documenting.

For example, we could express the `vector` constructors from (count,value), (values…), and (first,last) unambiguously, similar to the `assign` overloads but able to act as constructors and using non-overloaded names to demonstrate how that would look:

```
template<class T, class Allocator>
class vector {
    // ...
public:
    void set_fill  ( in size_type count, in T value,    in Allocator alloc = Allocator() ) out;
    void set_values( in initializer_list<T> values,     in Allocator alloc = Allocator() ) out;
    template<class InputIt>
    void set_from  ( in InputIt first, in InputIt last, in Allocator alloc = Allocator() ) out;
    // ...
};
```

Even for `vector<int>`, the canonical confusing example, this makes calling code's constructors unambiguous and clear, without having to resort to using `{ }` sometimes and teaching the differences between `( )` and `{ }` including why they can give nonintuitive results):

```
vector v<int> = uninitialized;                  // v is uninitialized, no constructor is called
if (cond) {
    v.set_fill(3, 42);                          // v is constructed with values { 42, 42, 42 }
} else if (cond2) {
    v.set_values(3, 42);                        // v is constructed with values { 3, 42 }
} else {
    v.set_from(myset.begin(), myset.end());     // v is constructed
}
```

Note that, as in the previous section, `out` continues to give us the ability to directly call alternative constructors for a locally declared object.

And because `out` parameters can be written on non-member non-friend functions, users can write their own constructors that delegate to other constructors.

2.3 `= initializer` enables consistent and truly uniform initialization

Building on the above consistent uninitialization via `=uninitialized`, next we add consistent initialization via `=initializer`, and furthermore consistent value-setting for both initialization and assignment:

- works consistently for **all types** (not just PODs);
- works consistently in **declaration and assignment** syntaxes;
- allows all value-setting to be **consistently spelled as `=initializer`**;
- last but not least, lets us give **consistent guidance** for all value-setting in C++.

We allow

```
MyType var = initializer;
```

and

```
MyType var = uninitialized;
// ...
var = initializer;                  // definite first use ⇒ construction
// ...
var = initializer;                  // otherwise, assignment
```

where *initializer* may be:

- a single identifier or expression;
- an *initializer-list*; or
- a parenthesized list that is a parameter list to a constructor (possibly `operator=`, see §2.4*ff*).

and when the destination object is already initialized, the semantics for assignment are that an `operator=` taking *initializer* is selected if available, otherwise a value is constructed from *initializer* and move-assigned.

In each of the following examples, the first two sets of lines have identical semantics (i.e., call the identical functions with identical parameters). The third line demonstrates that the identical syntax works for assignment.

Value-setting from a single variable or expression is pretty much what we already have, expanded as above to allow deferring a constructor call via `=uninitialized`:

```
string s1 = other;                  // construct from other
string s2 = uninitialized;
s2 = other;                         // construct from other
s2 = other;                         // assign from other
```

Value-setting from an initializer list is pretty much what we already have, expanded again with `=uninitialized`:

```
vector<int> v1 = {1, 2, 3};         // construct from {1, 2, 3}
```

```
vector<int> v2 = uninitialized;
v2 = {1, 2, 3};                     // construct from {1, 2, 3}

v2 = {1, 2, 3};                     // assign from {1, 2, 3}
```

I propose also allow value-setting from a parenthesized list of arguments:

```
vector<int> v = (1, 2);             // construct from (1, 2)

vector<int> v = uninitialized;
v = (1, 2);                         // construct from (1, 2)

v = (1, 2);                         // assign from (1, 2)
```

Note that this proposes a minor breaking change, only after the token sequence `= (` , because today `(1, 2)` is valid using the comma operator. Such code is rare, and in common cases where the expressions have no side effects, such as just naming two variables or literals, today it is likely a latent bug that does not mean what the programmer intended. I argue that making `=` consistent outweighs backward compatibility with rare edge cases, and outweighs the consistency of having `(a,b,c)` mean the comma operator elsewhere but a list of values if preceded by `=`. Also, code that wants to maintain the current meaning can add another pair of `()`.

Note also that this enables us to express something we could never (easily) express before, namely "**default assignment**" to set an already-constructed value to a default state, which is asymmetrically missing in current C++ (allowed for construction but not for assignment):

```
vector<int> v = ();                 // default construction

vector<int> v = uninitialized;
v = ();                             // default construction

v = ();                             // default assignment (new: currently not allowed)
```

Note this already works for some types, including `vector`, using `{}` syntax:

```
v = {};                             // default assignment (already legal)
```

For backward compatibility with C++20 types that do not have the nullary assignment operator described in the next section, we can make this mean to invoke the default constructor followed by the move constructor.

2.4 Segue toward generalized `operator=`: Zero/multiple params

At this point, the reader might anticipate that this paper would additionally propose that we allow an assignment operator to be declared with no parameters or multiple parameters, as we already do for constructors, for example:

```
class X {
    // ...

public:
    X& operator=();                 // (nullary) default assignment operator

    X& operator=(int, string);      // assignment operator from (int, string)
};
```

and then by the prior rule in this section to prefer an assignment operator if available that will be preferred if present. This makes `operator=` consistent with constructors.

Yes, this paper does propose that. But we can be even more general (and efficient): Consider what happens if those are `in` parameters…

2.5 Generalized `operator=`, part 1: `in` "that"

`in` (§1.3.1) naturally gives the efficiency of copying from cheap-to-copy values without writing a by-value parameter, and otherwise from automatically moving from rvalue arguments without writing `const&`/`&&` overloads:

```
class X {
    // ...
public:
    X& operator=(in int i, in string s);   // i: by-value copy semantics
};                                         // s: by-reference as if const&/&& overloaded
```

What if the parameter of a copy constructor or copy assignment operator is declared `in`, which means "by value if cheap to copy, else as if with a `const&`/`&&` overload"? First, consider the cheap-to-copy case:

```
class Point {
    int x, y;
public:
    Point(in Point that);                  // that: by-value copy semantics
    Point& operator=(in Point that);       // that: by-value copy semantics
};
```

This performs efficient copying using the normal 'pass cheap-to-copy objects by value' rule we teach in general, but that until now we have not allowed for copying special member functions.

Second, and more importantly, note these also naturally move from rvalues when the type is not cheap to copy:

```
class X {
    vector<widget> vw;
public:
    X(in X that);                  // that: by-reference as if const&/&& overloaded
    X& operator=(in X that);       // that: by-reference as if const&/&& overloaded
};
```

This makes them naturally **unified copy and move** operations. This can make code clearer by helping to prevent needless code duplication between copy and move functions that otherwise have the same structure (read from source member values the same way), which not only keeps copy and move consistent but makes them easier to write and maintain. Here they are side by side:

C++20 equivalent	**This paper (proposed)**

`class X {`
`  // ...`
`public:`
`  X(const X& that);`
`  X(X&& that);`
`  X& operator=(const X& that);`
`  X& operator=(X&& that);`
`};`
	`class X {`
	`  // ...`
	`public:`
	`  X(in X that);`
	`  X& operator=(in X that);`
	`};`

Note This paper does not currently allow overloading on only the parameter passing style. A motivating example to allow overloading could arise here: If copy and move naturally perform work *in a way that is algorithmically different*, then we could have a motivating case to allow them to be written separately by overloading `in` and `move`. For example:

```
class X {
    vector<widget> vw;
public:
    X(in X that);                 // copy (could be copy and move if it were alone)
    X(move X that);               // move (explicit overload for move if desired)
    X& operator=(in X that);      // copy (could be copy and move if it were alone)
    X& operator=(move X that);    // move (explicit overload for move if desired)
};
```

Note that for `operator=`, like all member functions, the current default for `this` is non-`const` ⇒ `inout`.

Next, consider what happens when we make `this` an `out` parameter.

2.6 Generalized `operator=`, part 2: `out` "`this`"

We already saw that this proposal allows `this` to be an `out` parameter, with the semantics that when invoked with an uninitialized object the members are treated as if `=uninitialized` and therefore naturally must be initialized, in order, in the body (i.e., an `out` "`this`" function is inherently (also) a constructor):

```
class X {
    widget w;
    gadget g;
public:
    X& operator=(/*...*/) out {
        w = /*...*/;
        g = /*...*/;              // note: must init g after w (compiler enforced)
        return *this;
    }
};
```

This is a **unified construction and assignment** function. For example:

```
X x1 = /*...*/;                   // calls operator=, constructs x.w and x.g

X x2 = uninitialized;
x2 = /*...*/;                     // calls operator=, constructs x.w and x.g

x2 = /*...*/;                     // calls operator=, assigns x.w and x.g
```

Note that the same `operator=` function is being called in each case. Of course, if construction and assignment naturally perform different work, they can be written separately as usual.

Note This function can behave as a constructor, and has the same efficiency as if using a mem-init-list, but no mem-init-list is required. Furthermore, this allows the flexibility of doing arbitrary work as needed before constructing a data member, including initializing data members along different paths as shown in §2.1. if that is useful, instead of being forced to first construct a data member with a dead-write placeholder value before we can calculate a real program-meaningful value in the

body. So this allows strictly more flexibility and expressiveness (and often efficiency) than constructors, with no special-purpose syntax unique to constructors.

This can make code clearer by helping to prevent needless code duplication between constructors and assignment operators that initialize their members the same way, which not only keeps construction and assignment consistent but makes them easier to write and maintain. Of course, if construction and assignment naturally perform different work, they can be written separately as usual.

Side by side:

C++20 equivalent	**This paper (proposed)**

`class X {`
`  // ...`
`public:`
`  X(/*...*/);`
`  X& operator=(/*...*/);`
`};`
	`class X {`
	`  // ...`
	`public:`
	`  X& operator=(/*...*/) out;`
	`};`

2.7 Together: Unified {copy,move}×{construction,assignment}

Finally, if we write `operator=` with both `out` "`this`" and taking a single `in` parameter of its own type we get a **single function that performs copy construction, move construction, copy assignment, and move assignment**:

```
class X {
    widget w;
    gadget g;
public:
    X& operator=(in X that) out {    // constructs *this if uninitialized, else assigns
        w = that.w;                  // moves from that if rvalue, else copies
        g = that.g;
        return *this;
    }
};
```

Side by side:

C++20 equivalent	**This paper (proposed)**

`class X {`
`  // ...`
`public:`
`  X(const X& that);`
`  X(X&& that);`
`  X& operator=(const X& that);`
`  X& operator=(X&& that);`
`};`
	`class X {`
	`  // ...`
	`public:`
	`  X& operator=(in X that) out;`
	`};`

Note This paper does not currently allow overloading on only the parameter passing style. A motivating example to allow overloading could arise here: If copy and move assignment naturally perform work *in a way that is algorithmically different*, then we could have a motivating case to allow them to be written separately by overloading `in` and `move`. See parallel Note in §2.5.

And this is (in my opinion, wonderfully) consistent, because now we can really reach that "`=` means setting the value" consistently throughout the language:

- **A class author can write `operator=` for all value setting**, both construction and assignment. And it's simple: By default they can conveniently write a single `=` function to enable copy construction, copy assignment, move construction, and move assignment, including naturally efficient move-from-rvalue semantics, but always with the option of writing separate functions if desired. In this proposal, there is now never a need to write a traditional constructor with its magic rules (e.g., can't have a return value, requires unique mem-init-list syntax), because every constructor can be written `operator=(/*...*/) out` with (a) equal or better efficiency in all cases, and (b) strictly better initialization flexibility and initialization guarantees.
- **A class user can write `=` for all value setting**, both construction and assignment. We can teach users simply that "`=` means `=`" — all operations invoked using `=` syntax (both construction and assignment) can invoke an `=` function in the class. There is no need to teach that "`=` in a declaration doesn't mean assignment and doesn't call `operator=`" because now it can.

Note It is high time for consistent initialization and assignment in C++. With this extension, we can (finally) teach a simple rule for value-setting in C++: Always write `= initializer`, it is as efficient on construction as if you had omitted the `=`, and it is consistent with assignment.

Consistent initialization and assignment has long been a thorny problem in C++, and many experts have attempted to improve the situation. The three features of (1) parameter passing, (2) initialization, and (3) generalized value setting are all interrelated, and in this proposal embracing

(a) the path-based concept of definite first use, and

(b) `out` parameters to let that first use extend through function calls

constitutes a fundamental (and, I argue, necessary and timely) change in approach that directly enables expressing this clean unification also of value-setting.

If we pursue this direction, we would obsolete 23+% of all published C++ "gotcha" guidance and may put at risk our language's hard-won reputation for arcane difficulty. But C++ still has other corner cases, so all is not lost; this paper only removes two of the in-practice-largest single sources of C++ usage and teaching complexity. Removing another 65% is the topic of other/future papers.

Appendix A: Implementation options for §1

Some parameter types pass additional information flags as summarized in §1.1.2.

Because this information is known at compile time at any given call site, we can choose to communicate it either statically or dynamically. This section describes those two major implementation options using the following base example, and a scalable dispatch mechanism for large numbers of `in` parameters all used with the same definite last use.

To show the two forms of `in`:

- `A` is small and trivially copyable, so the calling convention is to pass it by value.
- `B` is not, so the calling convention is to pass it by pointer.

We will consider this example:

```
void f( in A a, in B b, out C c, forward D d ) {
    // code section 1
    use(a);
    // code section 2
    use(b);
    // code section 3
    c = {1, 2, 3};
    // code section 4
    use(d);
    // code section 5
}
```

Notes:

- These are all concrete types for clarity. The implementation strategies are identical in a template.
- For `out` called with an uninitialized argument, if the callee constructed the argument then in the case where the callee exits via an exception the argument must be destroyed. It is cleanest to do this in the callee, so that's what the following code samples will show. An alternative, but *not* recommended, is to do the destruction in the caller, which would require extending the calling convention to return an additional `bool` whether the argument was constructed that each call site must test and conditionally perform the destruction, which is chatty and more complex and proliferates the check at each call site.

These can be passed statically or dynamically, as illustrated in the next sections.

A.1 Basic option 1: Static

To pass the information statically, the function would be compiled as if it had been written as:

```
template< bool     __b_arg_is_nonconst_rvalue,
          bool     __c_arg_is_uninitialized,
          typename TD,
          bool     __d_arg_is_uninitialized >
    requires std::is_same_v<TD, D>
void __generated_f( A a, const B* b, C* c, TD&& d )

try {
    bool __c_arg_was_constructed_by_f = false;

    // code section 1

    use(a);                                     // unchanged

    // code section 2

    // for initial exposition, this shows a brute-force "branchy" approach that would be
    // combinatorial for multiple parameters; see A.4 for a scalable efficient algorithm
    if constexpr( __b_arg_is_nonconst_rvalue )
        use(std::move(*b));                     // treat as non-const rvalue
    else
        use(*b);                                // treat as const lvalue

    // code section 3

    if constexpr( __c_arg_is_uninitialized )
        new (c) C{1, 2, 3} , __c_arg_was_constructed_by_f = true;
    else
        *c = {1, 2, 3};                         // or, if unvailable, c = C{1, 2, 3}

    // code section 4

    // + pass _d_arg_is_uninitialized to any 'out' or 'forward' callee parameter
    use( std::forward<TD>(d) );                 // forwarding pattern

    // code section 5

} catch(...) {
    if( __c_arg_was_constructed_by_f ) c->~C();
}
```

Notes Using `if constexpr` is more convenient and less combinatorial than spelling out specializations.

These could be `constexpr` parameters as proposed in [P1045R1], and be operationally equivalent.

A.2 Basic option 2: Dynamic

To pass the information dynamically, the function would be compiled as if it had been written as:

```
void __generated_f( A a, const B* b, C* c, D* d,
                    bool __b_arg_is_nonconst_rvalue,
                    bool __c_arg_is_uninitialized,
                    bool __d_arg_is_nonconst,
                    bool __d_arg_is_rvalue,
                    bool __d_arg_is_uninitialized )
try {
    bool __c_arg_was_constructed_by_f = false;

    // code section 1

    use(a);                                     // unchanged

    // code section 2

    // for initial exposition, this shows a brute-force "branchy" approach that would be
    // combinatorial for multiple parameters; see A.4 for a scalable efficient algorithm
    if( __b_arg_is_nonconst_rvalue )
        use(std::move(*b));                     // treat as non-const rvalue
    else
        use(*b);                                // treat as const lvalue

    // code section 3

    if( __c_arg_is_uninitialized )
        new (c) C{1, 2, 3} , __c_arg_was_constructed_by_f = true;
    else
        *c = {1, 2, 3};                         // or, if unvailable, c = C{1, 2, 3}

    // code section 4

    // for initial exposition, this shows a brute-force "branchy" approach that would be
    // combinatorial for multiple parameters; see A.4 for a scalable efficient algorithm
    // + pass _d_arg_is_uninitialized to any 'out' or 'forward' callee parameter
    if( __d_arg_is_nonconst )
        if( __d_arg_is_rvalue ) use(std::move(*d));
        else                    use(*d);
    else
        if( __d_arg_is_rvalue ) use(std::move(const_cast<const D&>(*d)));
        else                    use(const_cast<const D&>(*d));

    // code section 5

} catch(...) {
    if( __c_arg_was_constructed_by_f ) c->~C();
}
```

Note These can be packed into bitflags to make the stack frame smaller, which would be operationally equivalent. We should just measure which is faster, a slightly smaller stack frame or fewer bit-masking instructions. In this example, we could pass a single additional parameter of type:

```
enum __f_param_flags {
    __b_arg_is_nonconst_rvalue = 1,
    __c_arg_is_uninitialized = 2,
    __d_arg_is_nonconst = 4,
    __d_arg_is_rvalue = 8
};
```

A.3 Current approach: Static for `forward`, dynamic for `in` and `out`

This table summarizes the pros and cons.

	Option 1: Static	**Option 2: Dynamic**
Pros	No run-time overhead, including no branches	Can be an ordinary function: If `&f` works without this parameter passing style, it still works Minimum possible binary code space overhead: in many cases, even better than we can do right now by hand because today we must write overloads
Cons	Binary code space overhead, including code explosion: All the common `// code sections` are duplicated a combinatorial number of times (modulo COMDAT folding of identical bodies) Effectively every function with an `in`, `out`, or `forward` param becomes a template; and because `in` is the dominant case, this means *most* functions that use the new style will effectively become templates Can't take `&f`	1 hinted branch per overload candidate per definite last use of an `in` or `forward` param 1 branch per definite first use of an `out` param Changes the calling convention of the function, `&f` gives a different function types

Our current plan for initial prototyping is to use a variant of Option 1 (static) for `forward` flags (basically just directly automate today's forwarding pattern), and a variant of Option 2 (dynamic) for `in` and `out` flags.

This has user-visible semantic design consequences. We teach:

- The programmer can't take the address of an otherwise-ordinary function that has `forward` parameters; it is an implicit template. This is the same rule we teach for `auto` parameters.
- The programmer should prefer forwarders to be small functions, since they will be stamped out for each combination of parameter value category and const-ness.

We think "static implementation for `forward`, dynamic for `in` and `out`" is likely a sweet spot because forwarders already behave this way in today's C++, and `forward` still gives benefit because it is strictly more convenient.

So the function would be compiled as if it had been written something like this:

```
template <class TD>
    requires std::is_same_v<TD, D>
void __generated_f( A a, const B* b, C* c, TD&& d,
                    enum { __b_arg_is_nonconst_rvalue = 1,
                           __c_arg_is_uninitialized = 2 } __flags = 0 )

try {
    bool __c_arg_was_constructed_by_f = false;

    // code section 1

    use(a);                                     // unchanged

    // code section 2

    // for initial exposition, this shows a brute-force "branchy" approach that would be
    // combinatorial for multiple parameters; see A.4 for a scalable efficient algorithm
    if( __flags & __b_arg_is_nonconst_rvalue )
        use(std::move(*b));                     // treat as non-const rvalue
    else
        use(*b);                                // treat as const lvalue

    // code section 3

    if( flags & __c_arg_is_uninitialized )
        new (c) C{1, 2, 3} , __c_arg_was_constructed_by_f = true;
    else
        *c = {1, 2, 3};                         // or, if unvailable, c = C{1, 2, 3}

    // code section 4

    // + pass _d_arg_is_uninitialized to any 'out' or 'forward' callee parameter
    use( std::forward<TD>(d) );                 // forwarding pattern

    // code section 5

} catch(...) {
    if( __c_arg_was_constructed_by_f ) c->~C();
}
```

Because the `__flags` parameter defaults to `0`, this is equivalent to providing an additional overload without any flag parameters at all that is friendly to non-C++ FFI callers. For example, developers who write this:

```
void f( in A a, in B b, inout C c, out D d )
```

would get this:

```
void f( A a, const B* b, [[inout]] C* c, [[out]] D* d,
        enum { __b_arg_is_nonconst_rvalue = 1,
               __d_arg_is_uninitialized = 2 } __flags = 0 )
```

and implicitly also this too which is foreign language-friendly:

```
void f( A a, const B* b, [[inout]] C* c, [[out]] D* d )   // same, using flags==0 default
```

and so we now have a nice high-level way of authoring APIs that can be consumed by anything that understands C, plus automates the annotation distinguishing between `inout` and `out` parameters and enforces it to be now always correct (never gets out of sync with the source because it's language-enforced).

A.4 Multiple `in` and/or `forward` in the same definite last use

Summary: When multiple `in` and `forward` parameters have the same definite last use, we want to select the best match based on the actual arguments mutable-ness and rvalue-ness. However, do not want to implement a combination explosion of branches to test all possible argument flag combinations. Instead, observing that the overload candidate set is always the same, we want to implement a linear test of all possible overload candidates that differ in their mutable-ness/rvalue preference in those parameter positions.

> **Note** We only need to consider `in` and `forward`, because we never need to make a dispatch decision for `out`. For `out`, the only special rule is that the definite first use must be as the target of an assignment-expression or to pass it to an `out` parameter, both of which are static unconditional requirements regardless of whether or not the argument is initialized and so do not need to consult the flag to make a dispatch decision. So even if an `out` parameter appears in the same function call as the definite last use of one or more `in` and/or `forward` parameters, it has no effect except that if that same function call is also the definite *first* use of that `out` parameter we ignore any functions in the overload set that do not have an `out` parameter in that position (they are not viable).

For example, consider the following:

```
void f( in W w, in X x, forward Y y, forward Z z ) {
    g( w, x, y, z ); // definite last use
}
```

Consider an implementation that chooses the dynamic approach in A.2 for both `in` and `forward`, and generates:

```
void __generated_f( const W* w, const X* x, Y* y, Z* z,
                    bool __w_arg_is_nonconst_rvalue,
                    bool __x_arg_is_nonconst_rvalue,
                    bool __y_arg_is_nonconst,
                    bool __y_arg_is_rvalue,
                    bool __y_arg_is_uninitialized,
                    bool __z_arg_is_nonconst,
                    bool __z_arg_is_rvalue,
                    bool __z_arg_is_uninitialized ) {
    // call to g: what goes here?
}
```

It would not be scalable to implement a combinatorial set of branches for every combination of arguments. That brute-force approach would require 2^{I+2F} branches, where `I` is the number of `in` parameters and `F` is the number of `forward` parameters (the latter is doubled because it has two relevant flags). For example:

```
    // call to g: BAD brute-force implementation
    if( __w_arg_is_nonconst_rvalue && __x_arg_is_nonconst_rvalue &&
        __y_arg_is_nonconst && __y_arg_is_rvalue &&
        __z_arg_is_nonconst && __z_arg_is_rvalue ) {
        g( std::move(*w), std::move(*x), std::move(*y), std::move(*z) );
```

```
}
else if( __w_arg_is_nonconst_rvalue && __x_arg_is_nonconst_rvalue &&
    __y_arg_is_nonconst && __y_arg_is_rvalue &&
    __z_arg_is_nonconst && !__z_arg_is_rvalue ) {  // note: added "!" this time
    g( std::move(*w), std::move(*x), std::move(*y), *z );
}
else if( __w_arg_is_nonconst_rvalue && __x_arg_is_nonconst_rvalue &&
    __y_arg_is_nonconst && __y_arg_is_rvalue &&
    !__z_arg_is_nonconst && __z_arg_is_rvalue ) {  // note: other "!" this time
    g( std::move(*w), std::move(*x), std::move(*y), std::move(const_cast<const Z&>(*z)) );
}
// etc.
```

In the "BAD" brute-force implementation, using multiple `in` and `forward` parameters in the same definite last use function call expression causes a combinatorial explosion. Especially, multiple `forward` parameters are expected to be common, because of forwarded parameter packs.

Instead, note that for every argument combination:

- Name lookup is identical: We always find the identical candidate set.
- Only overload resolution differs: And then it depends only on whether a parameter to which an `in` or `forward` parameter is passed would have a better match based on mutable-ness or rvalue-ness.

So instead we should use the following algorithm, which generates a linear series of branches that is bounded to at most the number of viable candidates:

- Perform name lookup and get the set C of potentially viable overload candidates. (We can ignore non-viable candidates, such as candidates with the wrong arity or incompatible types.)
- For each candidate C_i in C, consider all parameters for which we are passing an `in` or `forward` parameter's definite last use as an argument, and compute the boolean condition B_i of mutable-ness and rvalue-ness flags that would make C_i the unambiguous better match.
- Sort C in order from most-restrictive to least-restrictive B_i, applying subsumption. If there exists any B_i and B_j for which neither is more-restrictive than the other, the call is ambiguous: Emit an error.
- For each candidate C_i in C, in order from most-restrictive to least-restrictive B_i, if B_i is nonempty then generate one `if`/`else` branch to test that combination of flags and invoke C_i appropriately. Note the last `else` condition should be empty (i.e., it is `else`, not `else if`).

In our example, let's say that name lookup for `g` finds four overloads, where `W` is implicitly convertible to `W2`:

```
void g( W  const&, X const&, Y const&, Z const& );  // g1
void g( W  const&, X &&,     Y const&, Z &&     );  // g2
void g( W  &,      X &&,     Y const&, Z &&     );  // g3
void g( W2 &,      X const&, Y const&, Z const& );  // g4
void g( std::string );                              // g5
```

Here:

- The set C is { `g1`, `g2`, `g3` }. Note that g4 is never viable because we cannot pass a `W` as the first argument, and g5 is never viable because it has the wrong arity.

- `g2` is a better match than `g1` when the second and fourth arguments are both non-const rvalues.
- `g3` is a better match than `g2` when the first argument is a non-const rvalue.

So we can emit the following linear generated code:

```
// call to g: max length of if/else chain depends only on arity of overload set,
//            independent of # in/forward parameters used

if( __w_arg_is_nonconst_rvalue &&
    __x_arg_is_nonconst_rvalue &&
    __z_arg_is_nonconst && __z_arg_is_rvalue ) {
    // invoke g3
    g( *w, std::move(*x), *y, std::move(*z) );
}

else if( __x_arg_is_nonconst_rvalue &&
         __z_arg_is_nonconst && __z_arg_is_rvalue ) {
    // invoke g2
    g( *w, std::move(*x), *y, std::move(*z) );
}

else {
    // invoke g1
    g( *w, *x, *y, *z );
}
```

Bibliography

[Brooks 1975] F. Brooks. *The Mythical Man-Month* (Addison-Wesley, 1975).

[ECMA-334] *Standard ECMA-334, C# Language Specification, 5th edition* (ECMA, 2017).

[lib-ext 2017-02-08] Discussion thread regarding making `std::move` implicit on last use of a path (WG21 email lists, 2017).

[P1045R1] D. Stone. “constexpr Function Parameters” (WG21 paper, 2019-09-27).

[P1179R1] H. Sutter. “Lifetime safety: Preventing common dangling (version 1.1)” (WG21 paper, 2019-11-22).

[Stroustrup 1994] B. Stroustrup. *The Design and Evolution of C++* (Addison-Wesley, 1994).

[Sutter 2014] H. Sutter. “Back to the basics! Essentials of modern C++ style” (CppCon, 2014).