Doc No.: D2288R0 DRAFT 2
Project: Programming Language - C++ (WG21)
Author: Andrew Tomazos <andrewtomazos@gmail.com>
Audience: EWG
Target: C++23
Date: 2021-01-21

Proposal of Designated Arguments

1. Abstract

Currently in C++, arguments are matched to parameters by type and position. We propose standardization of a language extension, called *designated arguments* (a.k.a. “keyword arguments”, “named parameters”), that enables arguments and parameters to be matched by name. We have addressed in the design all the issues raised regarding previous proposals of

this feature from 1992 to 2018. The breakthrough of our design is that it extends the function type system. A new notation opts each function parameter as one of positional, designator-optional or designator-required. In the latter two cases, the parameter name is exported as a significant part of its enclosing function type. This allows the feature to be successfully retrofitted onto existing C++ code, and enables a large range of important use cases - including overloading, overriding, forwarding, conversion and deduction of designators.

1.1 Informal Tour

We walk through gently and informally here the syntax and semantics of the extension. For detailed motivation, rationale, the underlying principles, requirements, design process, alternatives considered and so forth see sections 2 to 5. For the formal wording see section 7.

Parameters appear in function declarations. For example:

```
void f(int x) { /*...*/ }
```

This creates a new function of one parameter. The function has type `void(int)`.

Arguments appear in function call expressions. For example:

```
f(42);
```

This is a call to a function `f` with one argument.

The extension adds a new kind of argument called a **designated argument**. Existing arguments are called **positional arguments**.

So this:

```
f(42);
```

has a positional argument.

And this:

```
f(x: 42);
```

has a designated argument.

Designated arguments can also be written like:

```
f(.x = 42);
```

It means the same thing.

Likewise, designated initialization is also extended so that this:

```
struct S { int x; };
S s{.x = 42};
```

can also be written like this instead:

```
struct S { int x; };
S s{x: 42};
```

Often a designated argument has the same name as the value:

```
rotate(theta: theta);
```

When that happens you can omit the name like this:

```
rotate(:theta);
```

The extension adds two new kinds of parameters, called **designator-optional parameters** and **designator-required parameters**. Existing parameters are called **positional parameters**.

So this:

```
void f(int x);
```

has a positional parameter.

This:

```
void f(int. x);
```

has a designator-optional parameter.

This:

```
void f(int: x);
```

has a designator-required parameter.

Just remember: no dots is for positional, one dot is for designator-optional and two dots is for designator-required.

A positional parameter will only match with a positional argument:

```
void f(int x);

f(42); // OK
f(x: 42); // ERROR
```

A designator-optional parameter can match with either kind of argument:

```
void f(int. x);

f(42); // OK
f(x: 42); // OK
```

A designator-required parameter can only match with a designated argument:

```
void f(int: x);

f(42); // ERROR
f(x: 42); // OK
```

Designator-optional parameters only match with arguments in the correct position:

```
void f(int. x, int. y);

f(x: 42, y: 43); // OK
f(y: 43, x: 42); // ERROR
```

Designator-required parameters will match arguments in any position:

```
void f(int: x, int: y);

f(x: 42, y: 43); // OK
f(y: 43, x: 42); // OK
```

Designator-required parameters have to come last:

```
void f(int x, int. y, int: z); // OK
void f(int. y, int x, int: z); // OK
void f(int x, int: z, int. y); // ERROR
```

The kind of parameter and, in the case of the two new parameter kinds, the parameter name, become part of the functions type.

So this:

```
void f(int x);
```

has type `void(int)` as usual, which is the same as the type of:

```
void f(int y);
```

But this:

```
void f(int. x);
```

has type `void(int. x)`. This is a different type than that of:

```
void f(int. y);
```

which has type `void(int. y)`. It is also a different type than `void(int)`.

And similarly this:

```
void f(int: x);
```

has type `void(int: x)`. This is a different type than `void(int: y)`. It is also a different type than `void(int. x)`, and it is also a different type than `void(int)`.

As designator-required parameters can match arguments in any order, they are an unordered part of the function type. That is, their declaration order is not significant. This means that the function type of the following two functions is the same:

```
void f(int: x, int: y);
void f(int: y, int: x); // same as previous.  redeclaration.
```

ie The types `void(int: x, int: y)` and `void(int: y, int: x)` are the same type:

```
static_assert(
  std::is_same_v<
    void(int: x, int: y),
    void(int: y, int: x)
  >
); // OK
```

For the new parameter kinds, you can create a separate local name for the parameter with a slash, so this:

```
int f(int: really_long_name) { return really_long_name + 2; }
```

can be rewritten to:

```
int f(int: really_long_name/x) { return x + 2; }
```

The name on the left of the slash is the external name that is part of the function type and used by callers. The name on the right of the slash is a local name only visible in the function definition. So the function continues to be called like:

```
f(really_long_name: 42);
```

and continues to have type `int(int: really_long_name)`.

The slash can also be used to omit the local name to indicate the parameter is intentionally unused:

```
void f(int x) {} // WARNING: x unused

void f(int /*x*/) {} // OK

void f(int: x) {} // WARNING: x unused

void f(int: /*x*/) {} // SYNTAX ERROR

void f(int: x/) {} // OK
```

Because the parameter names are now part of the type, it means you can now overload on parameter name with the new parameter kinds:

```
void f(int. x) { cout << 1; }
void f(int. y) { cout << 2; }

f(x: 42); // outputs 1
f(y: 42); // outputs 2
```

Overload resolution works in the viability phase the way described before in terms of which kinds of arguments can match which kinds of parameters. Positional arguments are a viable match to both positional parameters and designator-optional parameters. Designated arguments are a viable match to both designator-optional parameters and designator-required parameters. Designated arguments can match designator-required parameters from any position (all other arguments and parameters have to match by position as usual).

In the ranking phase, a positional argument is a better match to a positional parameter than a designator-optional parameter. So these two overloads work:

```
void f(int x) { cout << 1; }
void f(int. x) { cout << 2; }

f(42); // outputs 1   (better match)
f(x: 42); // outputs 2  (first not viable)
```

A designated argument is not a better match to either a designator-optional or a designator-required parameter:

```
void f(int. x) { cout << 1; }
void f(int: x) { cout << 2; }

f(x: 42); // ERROR: ambiguous
```

Designator-optional parameters have the same default argument rules as positional parameters, however, one of the cool things about designator-required parameters is that they work much better with default arguments.

If a function has any designator-required parameters, then only they may have default arguments:

```
void f(int x, int: y = 43); // OK
void g(int x = 42, int: y = 43); // ERROR
void h(int x = 42, int: y); // ERROR
```

But any subset of them may have default arguments. You don't have to give default arguments to only the last ones:

```
void f(int: x, int: y); // OK
void g(int: x = 42, int: y); // OK
void h(int: x, int: y = 43); // OK
void i(int: x = 42, int: y = 43); // OK
```

And because they can accept arguments in any position, this means when you call them you can omit any subset of them:

```
void f(int: x = 42, int: y = 43, int: z = 44) {
    cout << x << y << z;
}

f(); // outputs 42 43 44
f(x: 0); // outputs 0 43 44
f(y: 0); // outputs 42 0 44
f(z: 0); // outputs 42 43 0
```

```
f(x: 0, y: 1); // outputs 0 1 44
f(y: 1, z: 2); // outputs 42 1 2
f(z: 2, x: 0); // outputs 0 43 2
f(x: 1, y: 2, z: 3) // outputs 1 2 3
```

Designated arguments can be forwarded by all the usual suspects:

```
struct S { S(int: x); };

auto p = std::make_unique<S>(x: 42); // OK

std::vector<S> v;
v.emplace_back(x: 42); // OK
```

And this also works with designated initialization:

```
struct S { int x; };
auto p = std::make_unique<S>(x: 42); // OK
```

It works by a new mechanism that uses the new keywords **fwdarg** and **declarg**.

To write a new function that forwards a single argument with fwdarg you write:

```
void g(int x) { cout << 1; }
void g(int: y) { cout << 2; }

template<typename Designator, typename Arg>
void f(Arg&& fwdarg(Designator)/arg) {
    g(fwdarg(Designator): std::forward<Arg>(arg));
}

f(42); // outputs 1
f(y: 42); // outputs 2
f(z: 42); // ERROR
```

So to forward multiple arguments you write:

```
template<typename... Designators, typename... Args>
void f(Args&&... fwdarg(Designators)/args) {
    g(fwdarg(Designators): std::forward<Args>(args)...);
}
```

The way to think about it is that fwdarg stores and loads a parameter kind and name at compile-time to and from a type template parameter. When used in a parameter it stores it (deduces it). When used next to an argument it loads it. The types that are loaded and stored

are called **designator types**. They represent the parameter kind and parameter name of a parameter.

You can also use fwdarg to decompose and reassemble the parameter kinds and parameter names of a function type:

```
void g(int x, int. y, int: z) {
   cout << x << y << z;
}

template<typename T> struct S;

template<typename... Designators, typename... T>
struct S<void(T... fwdarg(Designators))>  {
    static void f(T... fwdarg(Designators)/args) {
       g(fwdarg(Designators): args...);
    }
};

S<void(int, int. y, int: z)>::f(42, y: 43, z: 44) // out 42 43 44
```

This is the same technique that the extension uses to make `std::function` work with designated arguments:

```
std::function<int(int: x)> p = [](int: x) { return x + 2; };

p(x: 42); // returns 44
```

The declarg keyword works similar to decltype. declarg is used to get the designator type (representing the parameter kind and parameter name) directly from a parameter. For example:

```
void f(int. foo) {
    using ParameterType = decltype(foo);
    using ParameterKindAndName = declarg(foo);
}
```

declarg is used in a variety of generic programming use cases involving designator types. A common one is forwarding without using an explicit parameter list and designator deduction:

```
auto lambda = [](auto&& fwdarg(auto)/arg) {
    f(fwdarg(declarg(arg)): std::forward<decltype(arg)>(arg));
};

lambda(x: 42);  // calls f(x: 42)
```

That concludes a basic tour of the functionality of the extension.

2. Motivation

Parameterizing functional interfaces, and passing arguments to the resulting parameters, have been universal and fundamental programming activities for over 60 years. Contemporary demonstration of this in C++ can be easily shown by searching for occurrences of opening parenthesis in ACTCD19 (codesearch.isocpp.org) and hand-classifying a random sample of the 352 million hits. An average source file contains over a hundred argument lists or parameter lists. There is almost one such list on every other non-blank line. Parameterization and argument passing are paradigm-independant. They occur in procedural, object-oriented and functional styles. They are used just as extensively across every C++ domain.

Because of this ubiquity - considering improvements to parameterization and argument passing should be a top WG21 priority.

2.1 Positional Parameters

In many cases the type of an argument or parameter is not sufficient to identify it. Some types, such as the fundamental types (bool, int, float, char, etc) are not specific enough to describe the semantics of an argument or parameter with respect to its enclosing function call or function definition. Furthermore, for functions of multiple parameters, several parameters may have the same or similar types.

Because of that, in addition to matching by type, arguments are matched to parameters by their position. This scheme was inherited from C and earlier languages, likely originating in mathematical functions that are defined in terms of (ordered) cartesian products of sets.

2.2 Argument Readability

A positional argument is just an expression, and so can be something simple like an identifier (“id expression”), or a literal, or can be an arbitrarily complex compound algebra. Usually the reader of an expression can at least deduce its type - and sometimes more about it from its tokens, spelling and the surrounding context - but many times the reader cannot deduce the full meaning of an argument in relation to the function call, even when the function has only one argument.

Consider, for example, what is the value of the following expression?

```
std::vector<int>(3,4)
```

Does this create a vector of two elements with values 3 and 4 ? Does this create a vector of three elements with values 4, 4 and 4 ? or does this create a vector of four elements with

values 3, 3, 3 and 3 ? It doesn't matter what the answer is: the point is that it is hard to tell the meaning of the arguments without memorizing the 10+ constructors of std::vector. This example is not cherry-picked - quite the opposite: If std::vector, which is one of the most used entities in C++ can be hard to read, what hope does a function or constructor have from a new domain-specific API?

2.3 Surrogate Parameter Order Problem

As most sets of parameters have no natural order, in the usual case, a surrogate order must be invented by the API designer. By definition, a surrogate order cannot be intuited, so API users must be burdened to learn and remember these orders as they read and write function call expressions.

2.4 Transposition Errors

Where the types of two positional parameters are the same or similar, this can lead to transposition errors, a common source of runtime bugs: eg. Writing `f(width,height)` instead of `f(height,width)`, or visa versa. As can be seen with this simple example, such errors can occur even with functions of as little as two arguments.

2.5 Ambiguous Overload Resolution

Even using both type and position, there may be several members of a function overload set that are ambiguous for some argument lists. As pointed out by [Brown 2018]/2.2, this comes up a lot in constructor design, where each class has a single overload set of constructors (because constructors all have to have the same "name" as the class). Various contortions are needed by API designers to make positional overload sets less ambiguous. This effect is even demonstrated throughout the C++ standard library itself.

2.6 Optional Parameters and Position Collision

Like many languages, C++ has a feature that enables parameters to be declared optional and given a default value ("default arguments" [dcl.fct.default]). Arguments to such parameters may be omitted, and the default value is used instead. This interacts poorly with matching arguments to parameters by position - as when omitting an argument it messes up the position of latter arguments - so only a contiguous suffix of an argument list can be omitted and defaulted. Apart from being an arbitrary restriction, this also makes extending a function with a new parameter problematic as outlined by the use cases walked through by [Hardgrave 1976].

2.7 Matching Arguments By Name

The idea of overcoming these problems by matching arguments to parameters by name was first recommended by [Hardgrave 1976]. Hardgrave drew inspiration from what we'd now call scripting languages - that were already using the technique in the 1960s and 1970s.

It solves all the problems with positional arguments:

- READABILITY: There is an immediately obvious benefit to readability of some function call arguments when marked with a name. The name serves to document the relationship between the argument and function, removing a reading dependency on looking up the parameter name and surrogate order. (see 2.2 and 2.3).
- SAFETY: Matching by name is immune to transposition errors. (see 2.4)
- DISAMBIGUATION: The names can be used to inform overload resolution, curing some ambiguities that would otherwise arise, and can also cross-check that the overload selected is the intended one. (see 2.5).
- DEFAULT ARGUMENTS: In cases where such a system allows ignoring of argument position, there is an added benefit that an arbitrary subset of defaulted arguments can be omitted, rather than a contiguous suffix, solving the collision problem (see 2.6).

2.8 Existing Workarounds in C++

Numerous workarounds are in existing practice to address these issues. We compare them to the proposed extension.

2.8.1 Comment Parameter Names: `f(/*width=*/1920, /*height=*/1080)`

Many coding styles encourage commenting names next to arguments where it is unclear what an argument means.

For example [Naumann 2018] presents this function call from LLVM:

```
File = HS->LookupFile(
   InputFile,
   SourceLocation(),
   /*isAngled=*/false,
   /*FromDir=*/nullptr,
   /*CurDir=*/UnusedCurDir,
   Includers,
   /*SearchPath=*/nullptr,
   /*RelativePath=*/nullptr,
   /*RequestingModule=*/nullptr,
   /*SuggestedModule=*/nullptr,
   /*IsMapped=*/nullptr,
   /*SkipCache=*/true);
```

There even exists some tooling to help check that these comments are correct. Although, the limited efficacy of these tools, without some standard syntax to operate on, could explain their lack of wide-spread adoption.

If we alter this scheme to make the comments into real syntax:

```
File = HS->LookupFile(
   InputFile,
   SourceLocation(),
   isAngled: false,
   FromDir: nullptr,
   CurDir: UnusedCurDir,
   Includers,
   SearchPath: nullptr,
   RelativePath: nullptr,
   RequestingModule: nullptr,
   SuggestedModule: nullptr,
   IsMapped: nullptr,
   SkipCache: true);
```

and then we check them with tooling against parameter names, we arrive at what is essentially the proposal of [Naumann 2018].

While such a scheme does solve 2.2 and to some extent 2.4 (putting aside for the moment the safety concerns of [Gibbons 1992]), it violates requirements 3.2, 3.3, 3.5, 3.6 and 3.7, and it does not solve problems 2.3, 2.5 or 2.6.

2.8.2 Wrap Parameter Set in Class Type: `f(options)`

Rather than having a long and complicated parameter set, another approach advocates wrapping the parameter set in a class type, and then using encapsulation and class invariants to validate arguments. The API user constructs the class type, mutates it as appropriate, and passes it to the function.

There is no good way to enforce statically arguments that are both required and designated by this method. One can describe required parameters as parameters of the Options class constructor - but they are then again positional parameters - with all the listed problems still applying apart from 2.6.

The solution also creates extensive boilerplate that obfuscates the functional interface. The user must study the interface/documentation of the provided Options class - rather than just viewing the parameter list.

2.8.3 Use Designated Initialization as Proxy: `f({.width=1920, .height=1080})`

This is a new technique available as of C++20. It is similar to 2.8.2, but rather than a fully-featured class type, the parameter set is replaced by a simple struct (aggregate class type), and then designated (aggregate) initialization is used to pass the parameters.

<TODO: Further analysis of this option>

2.8.4 Library-Based Solutions

There are numerous designated argument library solutions that exploit (some would say abuse) various core language features in weird and wonderful ways to approximate the proposed extension, but none have been widely adopted. For a survey of some of them we refer you to [Arena 2014].

In general, library-based solutions cannot compete with core language extensions, as the latter are strictly more powerful. Library-based solutions are almost always compromised by the restrictions placed on them by the core language - and that certainly is the case for keyword argument libraries.

But, we cannot standardize a core language extension for everything, as it would make the core language too complex for users and place too much of a burden on implementers.

The question then becomes, is a designated arguments extension worth the complexity and burden of a core language extension over a library-based solution?

Given the prevalence of arguments and parameters in code (see introduction to 2.) - we answer in the affirmative. Anything that can improve argument passing easily warrants core language extension - and, in fact, we even argue, designated arguments should be our top priority.

2.9 Criticisms

There are three main sources of criticism of previous proposals of this feature in C++. They are [Gibbons 1991], [Stroustrup 1994] and [EWG 2014]. [Ballo 2014] and [Naumann 2018] also tried to preempt a few issues. We aggregate them all in 2.9 and analyze them against this proposal.

2.9.1 Complexity

C++ is already quite a complex language, in particular overload resolution is quite complicated. Why do we want to increase its complexity with new kinds of arguments and parameters? Is it worth it?

The overall complexity of this extension is no greater than that of any of the major extensions added to C++ in the last decade (eg. move semantics, lambdas, coroutines, etc), and because of the ubiquity of argument passing (see introduction to section 2), this feature has more tangible application in real-world code than any of them. If the extension is of even minor net benefit to argument passing, multiplied across this massive application space, the feature easily carries its own weight in total benefit. The feature only needs to be learned once, and the knowledge will then be reused constantly and frequently throughout all applications of C++.

2.9.2 Long Parameter Lists

[Gibbons 1992] claims that “[Keyword arguments] are much more important for functions which take a large number of arguments. For functions with few arguments, the nature of the arguments is usually obvious from the name and purpose of the function call.”

Following this and referring to this [Stroustrup 1994] states that “[Keyword arguments] would encourage programming styles that ought not be encouraged.”, meaning the style of having overly long parameter lists. In [EWG 2014] this was further clarified with an example of a function with 34 parameters.

[Ballo 2014] responds with two points. The first is that many existing positional APIs have long parameter lists, and “making it easier to deal with such functions would solve a real problem and be materially useful” and the second point is similar to 2.2: “Even for functions with few arguments, when reading a call site one has relatively little information about the roles of the arguments.”

[Naumann 2018] attempts to mitigate the criticism by taking the tack that: “A readability improvement in one part of C++ can mean that developers will invest that bonus to make code less readable in other places.”

We’ve analyzed these arguments extensively, and we think we can settle this if we answer two questions:

Q1. Are keyword arguments useful for short parameter lists?

Q2. Do keyword arguments encourage long parameter lists?

We can answer Q1 by taking a random sample of function call expressions with short argument lists, and testing whether or not they would be more readable if their arguments had designators.

After some empirical investigation along these lines, it suggests an answer of yes. Some functions of even just one parameter can benefit from designation. At first blush this result can seem surprising - but upon further reflection studying these real-world concrete cases, we think we can explain them with the following insight:

For a given function, the set of potential parameters it could have is usually much larger than the parameters it actually does have. The parameters it actually has express how the function can be varied by the user - whereas the unrealized potential parameters, that are left out of the parameter list, express the functions behaviour that is kept fixed and constant. (This is like the difference between an approximate physics simulation vs the real underlying physical process.)

The API designer culls these potential parameters to control complexity - to keep the API simple - by eliminating configuration that is not relevant to existing API use cases (or that simply noone

has requested yet). So it stands to reason that the result of that decision (as to which subset is kept) is, many times, not obvious to API users. One would need to know the whole history of the function and its uses to guess. That is, an API user can know what a function does, and that it has one parameter, but still not know which parameter (of the dozens of potential parameters) the API designer selected to allow the functions behaviour to vary on. A keyword argument helps identify which one the API designer selected.

We can answer Q2 by surveying code in languages like C# and Python that have a keyword arguments feature, and comparing them to code surveys of languages without keyword arguments, like C and C++. If the average parameter list length is longer in languages with keyword arguments than languages without them, we can reasonably extrapolate that keyword arguments encourage long parameter lists. If they are the same, we can extrapolate that they do not.

We decided to focus our investigation on C# as it is more similar to C++ than Python. Python has dynamic typing which confounds fair comparison. (Python parameters do not have a fixed type, and will bind to an argument of any type - so Python APIs lean more heavily on keyword arguments.)

After surveying numerous C# projects, we observe no significant difference in parameter list length in C# vs C++. After some analysis we explain this result as follows:

While keyword arguments make functions with long parameter lists more usable, there are other stronger forces that limit parameter list length than non-availability of keyword arguments.

The first is the complexity of the function definition. Having a function with a large number of parameters means that you have a large number of variables (parameters are a kind of local variable) in the scope of a function definition, and this quickly leads to a mess that programmers tend to break up through various means (hierarchical organization of data or objects, breaking up large functions into smaller ones, factoring into configuration classes, inheritance, etc). Keyword arguments have no bearing on this effect - these reactions result in keeping parameter lists short regardless.

The second is that it is a well-known general best practice to keep things a reasonable length in code. Alan Perlis' Epigram 11 (Perlism #11) stated in 1982 that *"If you have a procedure with ten parameters, you probably missed some."*. In general it is understood that having long flat lists of anything leads to disorganization. Whether they be parameters, data members, statements, or any kind of syntactic construct really. The language offers numerous features to organize such complexity to make it more manageable (one of the primary benefits of high-level programming languages). Programmers do not suddenly forget about the benefits of that best practice because of keyword arguments.

We therefore conclude that the [Gibbons 1992] and [Stroustrup 1994] position on this issue, while a reasonable hypothesis to make, turns out to be false in both claims. Keyword

arguments are useful independent of parameter list length, and keyword arguments do not encourage long parameter lists.

3. Requirements

We list and justify the design principles, use cases and requirements upon which our design is based.

3.1 Backward Compatible

As per almost all new C++ features, it shall not make any changes that break existing code. Existing programs must keep working after migration to a new C++ standard with no changes to their behaviour. The motivation for this is well-known and well-documented.

3.2 Existing Parameter Names Insignificant

Existing parameters in C++ can (and usually do) already have names, but parameter names are currently only significant locally to their enclosing declaration. (Contrast this with the public member names of a class type, which are significant outside their class definition.)

To change the rules to make parameter names significant outside of their declaration would, in effect, make their name part of the “interface” of their entity (function, constructor, function type, lambda). In agreement with [Gibbons 1992], [Stroustrup 1994] and [EWG 2014] - we think such a retroactive change would be unacceptable.

At the time they were written, existing parameter names were not important in that fashion. Had they been significant at the time they were written, in many cases parameter names would have been selected differently, perhaps with a different naming scheme, or lengthier, with more care or with other unforeseen differences.

Also, it is possible (and a common occurrence) to have two declarations of the same function to differ in parameter names. [Stroustrup 1994] describes a coding style where this is actually used as a feature. If we made them a significant part of the functions type, then this would be a breaking change, as two such declarations would now declare two different functions, violating requirement 3.1.

There are concerns about this even beyond retroactivity. If we force all parameter names to become part of the interface of functions, this could encourage API designers to comment out parameter names. Common and standard APIs would now have to agree on parameter names, giving even more to have to agree on.

For these reasons we require that existing parameters are of a kind that has the same semantics as parameters today, that are matched by type and position, but not by name. We

rename this kind of parameter a “positional parameter”, and they have the same syntax and semantics as existing parameters. New kinds of parameters, with different syntax and semantics, will be added to support the proposed feature.

This requirement has been described by some as **“opt in”**. By using the syntax of the new kinds of parameters, you are “opting in” to having designated arguments passed to your API.

3.3 Subsumption of Designated Initialization

A “simple struct” (formally an aggregate type of class type) today can be subject to aggregate initialization, and as of C++20, this can be designated initialization (eg. `S{.x=42, .y=43}`). The possibility of such aggregate initialization is, in fact, almost the only reason we categorize types into aggregate and non-aggregate.

From a language design point-of-view, we consider aggregate initialization to be like an aggregate has an implicitly generated memberwise constructor. (At one point during the design of aggregate initialization, it was even considered to make that how the formal wording worked.)

There are some generally significant differences between aggregate initialization and what is possible via a user-defined constructor, however, for the purposes of this proposal they are not relevant.

Given this model of aggregate initialization being like a memberwise constructor, we require that the designated arguments feature should be compatible and uniform with designated initialization. For example, ideally we need to consider the use cases of taking an aggregate class, and adding a user-defined constructor (making it non-aggregate), but having its existing designated initialization continue to work with the keyword parameters of the new constructor. This is the same relationship (through uniform initialization) that positional aggregate initialization has with list initialization through a positional constructor.

In short: Designated arguments should be a superset of designated initialization. The two features should be uniform.

3.4 Designator-Safety Principle

There is a massive amount of existing production C++ code in the world, with estimates ranging from 10-100 billion lines. All of that code currently uses positional arguments and positional parameters. We have to be very careful that migrating existing code to designated arguments is safe (not error-prone).

For this reason we specify the designator-safety principle which states:

If a program is modified by adding a designator to an argument - and neither the original nor modified program are ill-formed - then both programs should have the same behavior.

This means that you can safely add a designator to a positional argument. If it was a mistake, then the error will be caught at compile-time. A mistake won't silently change the behavior of the program and create a run-time bug.

For example, suppose there is a function call in a program:

```
prepare_layout(3);
```

and that the program is compiling successfully.

Thinking that the parameter being set is called `nwidgets`, the API user then modifies this function call as follows:

```
prepare_layout(nwidgets: 3);
```

and the program recompiles successfully with this change (ie no compiler error). If the designator-safety principle is observed by the feature - then this code change has not changed the program's behavior.

After careful analysis of application of this principle, we have concluded that it is only feasible and practical for the implementation to enforce it on a single overload. Therefore, the burden of enforcing it over an entire overload set must be passed from the language designer onto the API designer. Concretely, this means that adding a designator to an argument may cause a different overload to be selected, and it is up to the API designer to make sure that the two overloads do the same thing with the modified argument, as a matter of best practice.

3.5 Forwarding Designators

There are a large number of function templates in the standard library and user-code that "perfectly" forward argument lists to other functions. To name just a few, `std::make_unique`, `std::make_shared`, `std::vector::emplace_back`, `std::optional::emplace`, and so on. We require that there be a way to write such functions so that they forward on designators.

For the standard library, ideally, these functions should be modified to use the new designator-aware idiom, or if this is not possible for ABI backward compatibility, at least new designator-aware overloads should be added, or, failing that, a parallel set of designator-aware versions of these function templates should be added with some prefix/suffix on their function name.

3.6 Overloading on Designator

We require the ability to overload based on parameter designators. As mentioned in the Motivation section this comes up a lot in constructor design, where overloading is unavoidable. Many times the number, type and order of parameters is an insufficient basis to properly disambiguate two functions. Many overload sets are extremely crowded for this reason. The added ability to disambiguate using a designator is necessary and one of the prime motivations for this proposal.

3.7 Control to API Designer

As a general principle we think that where there are issues of control, flexibility and complexity and there is a choice between giving these to either the API user or API designer, we should favor the API designer. The reason being that the API designer is usually in a better position to make decisions about how the interface of the function (overload sets) they are writing should work.

Another reason is that a function is, generally speaking, written once and used many times. It is more efficient to do the additional work while configuring the function interface, rather than doing the additional work many times at the many function call sites.

3.8 Separability of External and Local Parameter Names

Unlike existing parameters, it is already clear that the names of parameters of the new parameter kind(s) will need to be significant outside of their declaration to support the other requirements.

There is a coding style described by [Stroustrup 1994] that has “long, informative” parameter names in the interface, but uses “short, convenient” parameter names in the implementation. Borrowing his example:

```
// declaration in header
void reverse(int* elements, int length_of_element_array);

// ...

// definition in cpp
void reverse(int* v, int n) {
    // ...
}
```

In order to continue to support such a style, we require that the new kinds of parameters have a convenient way to introduce a long external parameter name separately from a short local name.

However, as most of the time the two names will be the same, in this majority case, the syntax should not require repetition of the name. The syntax should default to them being the same.

3.9 Intentionally Unused Parameters

Existing parameter names can be omitted in function definitions to indicate that they are intentionally used:

```
void f(int x) {} // WARNING: x is unused

void f(int /*x*/) {} // OK
```

As per the first paragraph of 3.8, it is clear this isn't going to work as-is for the new parameter kind(s), as their names will be significant.

There exists two other techniques to indicate that a parameter is intentionally unused, namely [[maybe_unused]] and casting to void:

```
void f(int x [[maybe_unused]]) {} // OK

void f(int x) { (void)x; } // OK
```

The first technique is a little cumbersome and the second requires repetition of the parameter name.

We therefore require there to continue to be a convenient way to indicate that a parameter of the new kind(s) is intentionally unused (as there is for existing parameters).

4. Possible Future Requirements

We list some requirements that have been suggested, but that we do not recommend immediately supporting.

4.1 Converting Designators to/from Compile-Time Strings

The requirement here is to:

- be able to reflect an argument designator to a compile-time string holding the spelling of the identifier
- be able to reflect a parameter designator to a compile-time string holding the spelling of the identifier
- be able to reify a compile-time string to an argument designator with that identifier

- be able to reify a compile-time string to a parameter designator with that identifier

We consider the relationship of a designated argument/parameter to a function, to be similar to the relationship between a class member and a class type. As such, the feature that would allow you to reflect or reify one, should be able to do so with the other.

Therefore, this compile-time string conversion requirement should properly and consistently be part of the upcoming general reflection system being worked on by WG21.

While we require that this extension be forward-compatible with the future reflection system, we do not require the keyword arguments extension to be gated on the reflection system. They may proceed on independent schedules, and will be integrated as they both ship.

4.2 Designated Template Arguments

There is a symmetry in the language between the relationships of (a) function arguments, function parameters, function declarations and function call expressions; and (b) template arguments, template parameters, template declarations and template ids.

So it seems a natural extension of this proposal to include matching template arguments to template parameters by name, in much the same fashion as function arguments and function parameters. For the most part, the syntax and semantics proposed could be applied to the template grammar with little change.

What isn't clear at time of writing is whether the motivation is as strong for such an extension. From studying occurrences of `<` in ACTCD19 we conclude that template argument and parameter lists are about 10x less prevalent than function argument and parameter lists, and when they do occur the overwhelming majority are unary. Furthermore, as pointed out by [Naumann 2018] most template arguments are type template arguments, and type-ids are usually much easier to read than expressions because they are usually already identifiers.

They are still very common, so we do not conclude that we should never support such a feature, however our recommended approach is to reduce the initial scope to only support functions, as they are clearly a higher priority, and then based on the standardization experience for functions, we can look at adding keyword template arguments in a later proposal for a later standard cycle.

5. Design

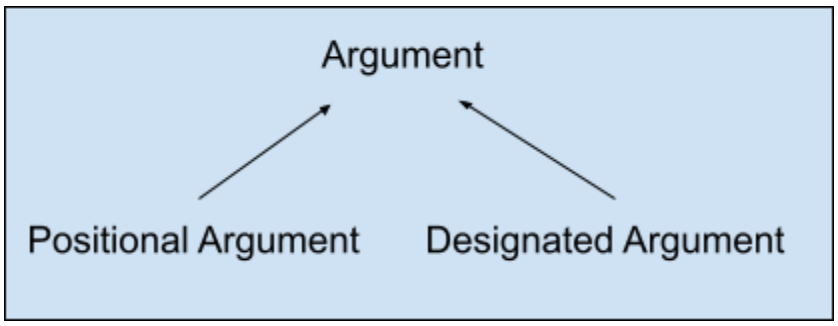
5.1 Arguments

Lists of comma-separated expressions can currently appear in numerous places throughout the grammar. We call these *arguments*.

We add to the grammar to allow some of these arguments to be marked by an identifier, we will call a *designator*.

Arguments not marked by a designator are called *positional arguments*.

Arguments marked by a designator are called *designated arguments*.



It follows from Requirement 3.1 that positional arguments must have the same syntax as existing arguments.

We now consider the different options for the syntax of a designated argument.

5.1.1. Designated Argument Syntax

5.1.1.1 Prior Art

[Hartinger 1991] proposed a new token `:=` and the syntax:

```
keyword-argument:
    identifier := expression
```

The motivation of this syntax was that it would not cause any syntactic problems or ambiguities, and it was felt to be “clearer” than the other options. We have not identified any syntactic problems or ambiguities with the other options, and we sent a request to EWG to look for people that felt this syntax was clearer than the alternatives, and we could find no-one that would advocate for it. So we have removed this syntax from consideration.

C++20 designated initialization uses the following syntax (abbreviated for exposition):

```
designated-initializer-clause:
    . identifier { /*...*/ }
    . identifier = { /*...*/ }
    . identifier = expression
```

The presence of the equals sign differentiates direct initialization from copy initialization. Argument passing is always copy initialization, so we should give special attention to that form.

Labelled statements in C and C++ use the following syntax:

labelled-statement:
identifier : statement

GCC, Clang and EDG accept both the C++20 form of designated initialization as well as an older deprecated form:

designated-initializer-clause:
identifier : expression

The above syntax was designed in 1991 as part of a GCC extension and then later changed in 1993 to the `.identifier = expression` syntax (deprecating previous). That syntax was standardized into C in WG14 N494 in 1995, shipped in 1999 C99, and then the C++20 designated initializer syntax is a subset of that C99 syntax.

N494 was based in part on a Fortran feature, and it is said that Ken Thompson originated the C designated initializer syntax in the late 1980s.

In Python keyword arguments have the syntax:

kwarg:
identifier = expression

This is possible in Python, because assignment is a statement and not an expression. In C++ this syntax would be ambiguous with an assignment expression.

In C# named arguments have the syntax:

named-argument:
identifier : expression

In the C++ named parameters proposals of 2010s, [Ballo 2014], [Naumann 2018] and [Brown 2018] the proposed keyword argument syntax is:

keyword-argument:
identifier : expression

Analyzing this we see three different options:

5.1.1.2 Option A1 - `f(.x = 42)`

Use a designated argument syntax that matches the standard designated initialization form:

designated-argument:
. identifier = expression

PROS:
- This satisfies Requirement 3.3, it is consistent with designated initialization so migrating between constructors and aggregate initialization is easy.

CONS:
- The period token is meant to look like member access. This is appropriate for designated initialization where the target is a data member, but for named parameters the target are parameters, so this no longer looks right.
- It's lengthier than other syntaxes, requiring two tokens instead of one.

5.1.1.3 Option A2 - `f(x: 42)`

Use a designated argument syntax of:

designated-argument:
identifier : expression

PROS:
- This syntax is consistent with labelled statements, the deprecated form of designated initialization, the C# syntax, and all of the C++ named parameters proposals of the last ten years (N4172, P0671, P1229).
- The colon is the more natural and intuitive syntax for labelling as it is rooted in natural language.

CONS:
- Violates Requirement 3.3, as it would be inconsistent with designated initialization.

5.1.1.4 Option A3 - both `f(.x = 42)` and `f(x: 42)`

Use a designated argument syntax of:

designated-argument:
. identifier = expression
identifier : expression

where both alternatives are equivalent, and extend

designated-initializer-clause:
designator brace-or-equal-initializer

identifier : initializer-clause

Such that the second alternative is equivalent to . identifier = initializer-clause. (This would also in effect undeprecate and standardize in C++ the existing extension of this second form of designated initialization.)

PROS:
- This has all the pros of both Option A1 and A2, in the sense that the natural syntax is made available while retaining compatibility with designated initialization as per Requirement 3.3.

CONS:
- Having two different ways of writing the same thing can create friction, and spawn arguments over style, like const east vs west, or int* p vs int *p, or class vs struct.
- Increases complexity as readers have to learn both syntaxes, and that they are equivalent.
- Has the largest syntactic footprint, taking up premium syntactic space we could use for other things.

5.1.1.5 Decision

We propose Option A3. While the Cons of that option are certainly not insignificant, we feel the "best of both worlds" Pros outweigh them:

- Style wars are to some extent unavoidable - tools (such as clang-format) are being actively developed that normalize such stylistic decisions in a configurable fashion.

- The increased complexity is somewhat mitigated by the fact that both syntaxes are fairly intuitive in their own right, even without background knowledge.

- Argument lists are such a prevalent part of C++ code - its hardly like we are proposing a large syntactic footprint for some esoteric or little-used feature. We should be liberal with the syntax in this case.

5.1.2 Implicitly Designated Arguments - `f(:x)` and `f(.=x)`

User experience in languages that have named parameters, and other similar systems, has shown that it is a common occurrence that when an argument is an expression consisting of a single identifier (an "id-expression"), the identifier of the expression is often the same as the designator. The reason why this happens is just common sense: the name given to a local variable (or an enclosing to-be-forwarded parameter) describes the meaning of that variable, and when it is passed as an argument, the meaning of the corresponding parameter is often the same, and so it ends up that they are named the same.

For this common situation we therefore propose that in such cases the designator may be omitted, and is taken implicity to be the same as the expression:

designated-argument:
identifier : expression
. identifier = expression
: identifier
. = identifier

For any identifier **X**, a *designated-argument* of the form `: **X**` or `. = **X**`, is equivalent to `**X** : **X**`.

For example:

`f(:x)` is equivalent to `f(x: x)`
`f(.=x)` is equivalent to `f(x: x)`

5.2 Parameters

Parameter lists appear in function types, function declarations, constructor declarations and lambda expressions.

Per Requirement 3.1 and 3.2 the syntax and semantics of existing parameters must remain unaltered, and we shall call them *positional parameters*. Positional parameters must continue to have insignificant names so may only match positional arguments.

We call the new kinds of parameters, as a group, *designatable parameters*.

A designated argument must be able to match a designatable parameter (for it to be otherwise, then they would be no different than positional parameters, and therefore pointless), but what about a positional argument? Can a positional argument match a designatable parameter?

Also, does a designated argument have to match a designatable parameter in the correct position? Or can they match from any position?

Clearly a designated argument in the same position as a designatable parameter must match (a designatable parameter that requires its designated argument to be in a different position would be nonsensical), but the combination of answers to the other question admits four potential subcategories of designatable parameter. Those being:

Designatable Parameter	Category 1	Category 2	Category 3	Category 4
..match designated argument in same position	YES	YES	YES	YES

..match designated argument in different position	NO	YES	NO	YES
..match positional argument in same position	NO	NO	YES	YES
..match positional argument in different position	NO	NO	NO	NO

So the design decision becomes: what subset of these four categories do we want to support?

We first note that we want to keep the extension simple, so the fewer categories the better.

Category 4 would cause violation of Requirement 3.4 (the Designator-Safety Principle). For if you removed a designator from an argument that matched a Category 4 parameter, the resulting positional argument may match up with a different parameter. This leaves Category 1, 2 and 3 remaining.

Category 1 is too constrained to be useful. Such a parameter would require both a correct designator and a correct position. We think this is overkill. We are convinced Category 1 does not carry its own weight, so in the interest of simplicity, we drop it. We are therefore left with Category 2 and Category 3.

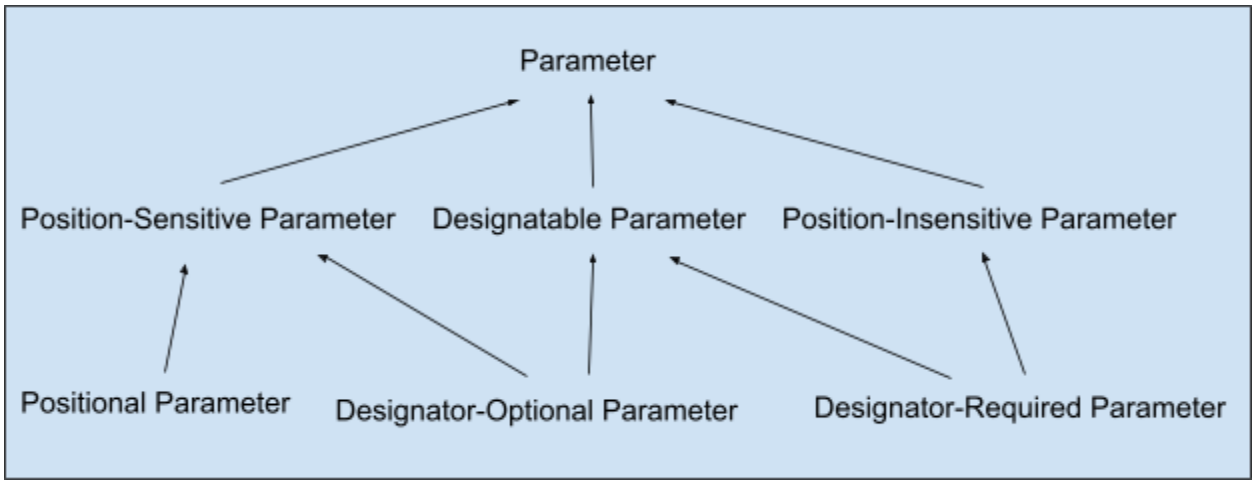
Category 3 works similarly to the C++20 rules for designated initialization. You must pass its argument in the correct position, but you may or may not designate the argument. We call this category a *designator-optional parameter*.

Category 2 requires a designated argument, for this reason we call it a *designator-required parameter*, but it may match a designated argument from any position.

Because both positional parameters and designator-optional parameters are sensitive to their arguments position (they only match arguments in the correct position), for convenience we call them collectively *position-sensitive parameters*.

It follows that designator-required parameters can also be called *position-insensitive parameters*.

Our taxonomy of the different parameter kinds is now complete:



While it would be technically possible to allow position-insensitive parameters to precede position-sensitive parameters, it would be quite confusing for position-insensitive things to affect the order of position-sensitive things. As is customary in other systems, we therefore require that arguments and parameters that participate in matching by position appear first in parameter lists and argument lists, and parameters and arguments that do not participate in matching by position, appear last.

This constraint can be achieved by requiring position-sensitive parameters to precede position-insensitive parameters in parameter lists - and so argument lists will also follow this arrangement without further force.

As previously stated, the syntax of positional parameters must be the same as current existing parameters.

We are left with designing a syntax to differentiate designator-optional parameters and designator-required parameters from each other and from positional parameters.

5.2.1 Designatable Parameter Syntax

5.2.1.1 Prior Art

C# has one kind of parameter, not three - so there is no special parameter syntax. This is partly because they implemented named parameters early in the languages life. Originally, the C# rule was that named arguments had to follow positional arguments, and all parameters were essentially Category 4. This reduces (but not eliminates) violations of the Designator-Safety Rule at the argument level, but it removes any flexibility from the function designer. An attempt was made to improve on this in C# version 7.2, whereby up to the last positional argument, named arguments are position-sensitive, after the last positional argument, named arguments are position-insensitive. We think this makes the rules about named arguments even more complicated for the function user. As per Requirement 3.7, our design has the flexibility (and

complexity) in the hands of the API designer. For example, for a given API a function designer, as a matter of style, may choose to only use only two, or even one, of the three different parameter kinds - which makes the API argument rules much simpler for the function users of that API.

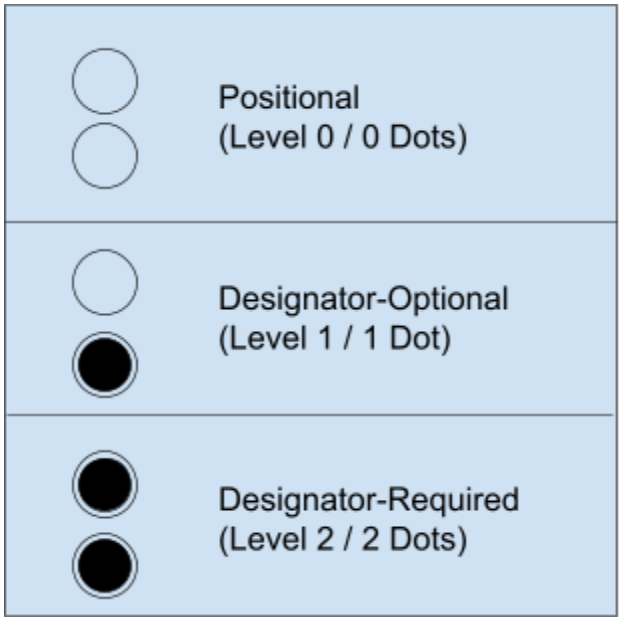
Python has two kinds of parameters, not three, but the two kinds are roughly similar to our designator-optional parameters and designator-required parameters. The syntax it uses is to place a token, specifically an asterisk `*` token, in the parameter list, this partitions the parameter list into designator-optional parameters and designator-required parameters.

5.2.1.2 Option P1 void f(int a, int. b, int: c)

This option introduces two new innermost declarators for function parameters, one for each of the two new kinds of parameter:

```
declarator-id:
   id-expression  /* positional parameter */
   ... id-expression  /* pack of parameters*/
   . identifier    /* NEW: designator-optional parameter */
   : identifier   /* NEW: designator-required parameter */
```

The visual idea here can be seen in the following diagram:



It also uses the same tokens as the argument syntax.

It’s also not a coincidence that the existing standard designated initialization syntax uses a period and the designator-optional declarator uses a period.

PROS:
- The indicator of the kind of parameter is local to the parameter declaration and context insensitive.
- The tokens are the same as those used for designated arguments.

CONS:
- There is no intuitive way to conclude which one means designator-optional and which one is designator-required without explanation. This would have to be learned (although it is easy to remember once learned from the above “two dot” chart)

5.2.1.3 Option P2 void f(int a, int? b, int! c)

This is similar to Option P1, but uses ? and ! instead:

```
declarator-id:
   Id-expression  /* positional parameter */
   ... id-expression  /* pack */
   ? identifier    /* NEW: designator-optional parameter */
   ! identifier   /* NEW: designator-required parameter */
```

Here the idea is like:

- you can maybe have a designator?
- you must have a designator!

PROS:
- The indicator of the kind of parameter is local to the parameter declaration and context insensitive.
- The difference between designator-optional and designator-required is easier to intuit from the tokens than the other options.

CONS:
- The tokens are quite visually “loud”. That is, the glyphs are big and complicated. Given how often they would be used, this is especially bad.
- The ! token can be confused with the meaning of the not operator.
- consteval was originally going to be spelled constexpr!, but we moved away from it for similar reasons.

5.2.1.4 Option P3 void f(int a, int: b; int: c)

In this option we use a single colon declarator for both designator-optional and designator-required parameters, and we partition them with a semi-colon:

```
declarator-id:
   id-expression  /* positional parameter */
   ... id-expression  /* pack */
```

: identifier / NEW: designatable parameter */*

parameter-declaration-clause:
parameter-declaration-list$_{opt}$ ...$_{opt}$
parameter-declaration-list, …
*parameter-declaration-list$_{opt}$; parameter-declaration-list /*NEW*/*

PROS:
- Uses a single token for all designatable parameters. This is visually simpler.
- The colon token is used in natural language to indicate labelling.
- Uses the same separator token, semi-colon, as the classic for statement and the in-condition declaration statement feature, and compound statements. All of which originated in natural language usage.

CONS:
- The indicator of the kind of parameter (at least designator-optional vs designator-required) is not local to the parameter declaration. One would have to look through the parameter list to find the semi-colon. Given a large number of parameters, this is problematic.
- In the common case of functions that have all designator-required parameters, especially in the C++ case where they get you position-insensitivity, this would require an initial semicolon after the opening parenthesis, eg `void f(;int:x)`, which reads a little awkwardly.

5.2.1.5 Decision

Weighing the Pros/Cons we think that Option P1 is marginally the best. We don't find huge differences between the Pros of the different options, and overall Option P1 seems to have the least Cons.

5.2.2 Split Parameter Names - f(int: x/y)

In order to support both Requirements 3.8 and 3.9, we create a syntax using the natural language meaning of the slash token to mean "alternative":

declarator-id:
id-expression
... id-expression
. identifier
: identifier
. identifier / identifier$_{opt}$
: identifier / identifier$_{opt}$

When a declarator-id introduces a parameter of a function definition, a declarator-id of the form **. X** or **: X** is equivalent to **. X / X** or **: X / X** respectively.

A declarator-id with a slash token (/) may only appear to introduce a parameter of a function definition. The first identifier **X** is the external parameter name exported to the function type, and the second identifier **Y** (if any) is the local parameter name that is only in scope for the function definition. If the second identifier is omitted, then no local parameter name is introduced. (This indicates that the parameter is intentionally unused)

So implementing the example from 3.8 with designatable parameters, it would read:

```
// declaration in header
void reverse(int*: elements, int: length_of_element_array);

// ...

// definition in cpp
void reverse(int*: elements/v, int: length_of_element_array/n) {
    for (int i = 0; i < n/2; i++)
       swap(v[i],v[n-i-1]);
}

// ...

// call to reverse
reverse(elements: &my_array[0], length_of_element_array: 10);
```

And the example from 3.9 would read:

```
void f(int: x) {} // WARNING: x is unused

void f(int: x/) {} // OK
```

This explicit omission of the local parameter name suppresses the automatic introduction of a local parameter name with the same identifier as the external parameter name, and indicates to the implementation that the parameter is intentionally unused.

5.3 Generic Designator Programming

In order to support Requirement 3.5 and related use cases, we propose a set of generic programming extensions, and prove their efficacy in implementations of std::make_unique, std::function and other applications.

The system was designed by looking for the minimal set of language extensions that enabled the use cases. We view these language features as an implementation detail that supports the requirements, rather than as a first class user-facing feature. (You can think of them as having

the same relationship to the proposal, as, say, reference-collapsing has in supporting move semantics.)

5.3.1 Designator Types

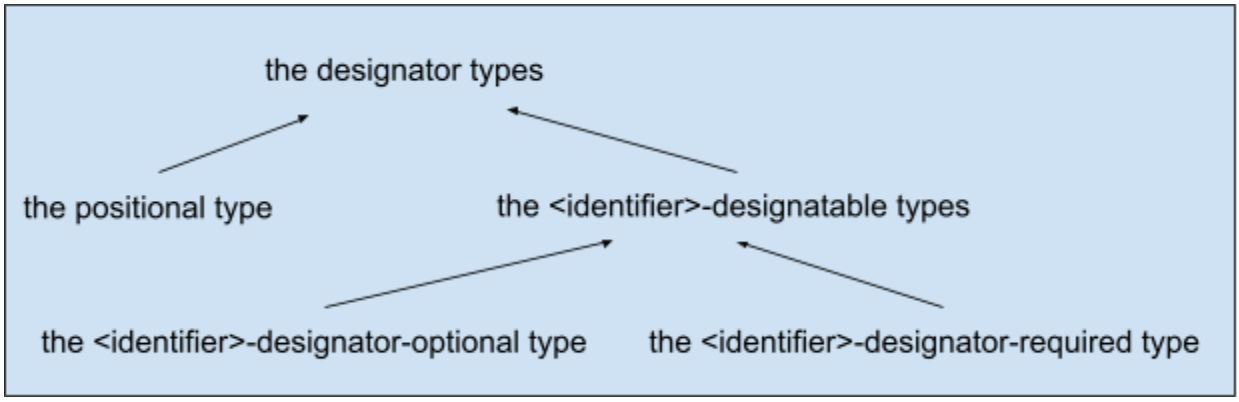
We introduce a family of built-in tag types called the *designator types*. These form a parallel hierarchy to the different parameter kinds and parameter names. Each designator type represents a combination of parameter kind and parameter name.

There is one designator type that represents all positional parameters, called “the positional type”.

For each possible unique identifier **X,** there are two designator types:

1. One representing all the designator-optional parameters that have parameter name **X** called “the **X**-designator-optional type”
2. One representing all the designator-required parameters that have parameter name **X** called “the **X**-designator-required type”.

Collectively, these two types are called “the **X**-designatable types”.



For example:

The parameter in the following function...

```
void f(int foo);
```

...is represented by “the positional type”.

The parameter in the following function...

```
void f(int. bar);
```

...is represented by “the bar-designator-optional type”. (This is one of “the bar-designatable types”, the other being “the bar-designator-required type”.)

The parameter in the following function…

```
void f(int: baz);
```

...is represented by “the baz-designator-required type”. (This is one of “the baz-designatable types”, the other being “the baz-designator-optional type”.)

5.3.2 `declarg` Specifier

We introduce a new full keyword **declarg**, and a new notation called the declarg specifier.

The declarg specifier takes a local parameter name and returns the designator type (5.3.1) representing that parameter:

simple-type-specifier:
  /*...*/
  declarg-specifier

declarg-specifier:
  declarg (identifier)

For example:

```
void f(int. foo/bar) {
    using T = declarg(bar);
}
```

T in the above is “the foo-designator-optional type”.

5.3.3 `fwdarg` Specifier

We introduce a new full keyword **fwdarg**, and a new notation called the fwdarg specifier.

The syntax takes a designator type as input, and produces an argument or parameter that corresponds to it. (It “forwards” the designator from the designator type - hence the name.)

The basic syntax is:

fwdarg-specifier:
  fwdarg (type-id)

The type-id shall be a designator type (5.3.1).

A fwdarg-specifier can be used to syntactically replace the identifier in an argument designator:

designated-argument:
identifier : expression
. identifier = expression
fwdarg-specifier *: expression*
.* fwdarg-specifier = *expression

designator:
. identifier
. fwdarg-specifier

designated-initializer-clause:
designator brace-or-equal-initializer
identifier : initializer-clause
fwdarg-specifier *: initializer-clause*

For example:

```
using T = /*...*/;

f(fwdarg(T): 42);
```

The argument produced by a designator-forwarder depends on the designator type:

- If the designator type is “the positional type”, then the argument produced is a positional argument;
- Otherwise, the designator type must be one of “the X-designatable types” for some X. In this case, the argument produced is a designated argument with identifier X.

For example:

```
void f(int foo, int. bar, int: baz) {
    using T1 = declarg(foo); // the positional type
    using T2 = declarg(bar); // the bar-designator-optional type
    using T3 = declarg(baz); // the baz-designator-required type

    g(
       fwdarg(T1): 42,
       fwdarg(T2): 43,
       fwdarg(T3): 44
    );
```

```
}
```

The function call to g is equivalent to `g(42, bar: 43, baz: 44)`

A fwdarg-specifier can also be used to syntactically replace the parameter kind and external parameter name in a parameter declaration:

declarator-id:
id-expression
... id-expression
. identifier
: identifier
fwdarg-specifier
. identifier / identifier$_{opt}$
: identifier / identifier$_{opt}$
fwdarg-specifier / *identifier*$_{opt}$

For example:

```
void f(int fwdarg(T)/x);
```

5.3.5 Deduction from arguments

For example, here is the real-world implementation of std::make_unique from libstdc++:

```
// TODAY: libstdc++ std::make_unique
template<typename _Tp, typename... _Args>
inline typename _MakeUniq<_Tp>::__single_object
make_unique(_Args&&... __args)
{
   return unique_ptr<_Tp>(
       new _Tp(
           std::forward<_Args>(__args)...
       )
   );
}
```

And here is how it is modified to make it designator-aware:

```
// PROPOSED: libstdc++ designator-aware version of std::make_unique
template<typename _Tp, typename... _Designators, typename... _Args>
inline typename _MakeUniq<_Tp>::__single_object
make_unique(_Args&&... fwdarg(_Designators)/__args)
```

```
{
   return unique_ptr<_Tp>(
        new _Tp(
            fwdarg(_Designators): std::forward<_Args>(__args)...
        )
   );
}
```

5.3.6 Deduction from function types

For example, here are the interesting parts of the class template specialization of std::function from libstdc++:

```
// TODAY: libstdc++ std::function:
template<typename _Res, typename... _ArgTypes>
    class function<_Res(_ArgTypes...)> {
    /*...*/
    _Res operator()(_ArgTypes... __args) const;
    /*...*/
}
```

And here is how it would be adjusted to make it designator-aware:

```
// PROPOSED: libstdc++ designator-aware std::function:
template<typename _Res, typename... _Designators, typename...
_ArgTypes>
    class function<_Res(_ArgTypes... fwdarg(_Designators))> {
    /*...*/
    _Res operator()(_ArgTypes... fwdarg(_Designators)/__args) const;
    /*...*/
}
```

6. Semantics

6.1 Function Types

In order to support Requirement 3.6 (Overloading) it is clear that the parameter kind and for designatable parameters, the parameter name, will have to become part of the function type.

Function types currently contain an ordered list of parameter types called the *parameter-type-list*. The parameter names are erased. These are what we now call positional parameters.

The parameter-type-list will need to be extended to keep for each parameter the parameter kind (positional, designator-optional or designator-required), and for designatable parameters (designator-optional and designator-required) the identifier.

To reflect this we rename it to the *parameter-set*.

It was considered whether or not this should only be done for designator-required parameters and that we could leave designator-optional parameters out of the function type. Such an approach would have the benefit that transforming a positional parameter into a designator-optional parameter would not be an ABI break (ie transforming a positional parameter into a designator-optional parameter would be a header-only change, and not require recompilation of the object file). Upon careful analysis we found numerous downsides of this approach:

- It would mean that you couldn't overload on designator-optional parameters. There are use cases in overload set design where the position-sensitivity of designator-optional parameters is used as a feature.

- It would break our current forwarding mechanism which is built atop designator-optional parameters. While this is secondary and we may be able to workaround an alternative, it certainly remains a con of this approach.

- It weakens linking of the parameter name across translation units, which partially violates Requirement 3.2. It would be more desirable for an incorrectly declared parameter name to be a hard error.

- By allowing an API user to give or change a designatable parameter name with a header-only change, it shifts control away from the API designer, which impacts Requirement 3.7. Ideally the provider of a precompiled library should be in control of which parameters are designatable and by what identifier.

- It would make designator-optional and designator-required parameters less consistent. There is an advantage to simplicity by having consistent rules that apply to all designatable parameters.

- If designator-optional parameters could be set without an ABI break, then they would have an advantage over designator-required parameters, even though the latter may be a better choice for a given API.

- The ABI break issue is mitigated by the possibility of introducing an overload with designatable parameters that delegates to the positional parameter overload.

For these reasons we decided to make both kinds of designatable parameters part of the function type.

Concretely, a parameter-set consists of two pieces:

(1) An ordered sequence of position-sensitive parameters. Each is either:

(a) a positional parameter, in which case the element is just a parameter type as usual; or

(b) a designator-optional parameter, in which case the element consists of an identifier and a parameter type.

(2) An unordered set of designator-required parameters. Each element consists of an identifier and a parameter type.

Two function types are different if they have different parameter-sets.

More precisely, two function types are different:

- if the ith position-sensitive parameters differ in parameter type (as usual); or in kind; or if they are both designator-optional parameters and they have different identifiers; or
- if they differ in designator-required parameters. That is, one contains an identifier the other doesn't - or, if not, a pair of parameters of the same identifier differs in parameter type.

6.2 Overload Resolution

6.3 Order of Evaluation

6.4 Mangling Extension

Mangling is out-of-scope of the standard, but we specify here for reference the concrete extension to the **C++ Itanium (Linux) ABI.** Other ABIs (such as MSVC) can follow in a similar fashion.

In general, function type mangling (and function mangling) is extended such that the new parameter kinds, and their identifiers, are added to the mangled parameter lists.
Designator-required parameters are sorted in identifier-order:
- A positional parameter consists of just a type as usual.
- A prefix character introduces a designator-optional parameter, followed by a name, and then a type
- A separator character indicates the start of the designator-required parameters.

<TODO>

7. Implementation

A complete implementation of this proposal is under development in one of the major C++ implementations. We intend proposing its merge and ship as a non-standard extension prior to pursuit of standardization.

8. Wording

TODO

[gram]

parameter-declaration :
  attribute-specifier-seq$_{opt}$ attribute-specifier-seq opt
  attribute-specifier-seq opt
attribute-specifier-seq opt
decl-specifier-seq
decl-specifier-seq
decl-specifier-seq
decl-specifier-seq
declarator
declarator = initializer-clause
abstract-declarator opt
abstract-declarator opt = initializer-clause

[functional.syn]

namespace std {
  template<class… Designators> struct designator_list_t {};

  template<class F, **class… Designators**, class... Args>
  constexpr invoke_result_t<F, **designator_list_t<Designators...>**, Args...> invoke(F&& f, Args&&... fwdarg(Designators)/args)
  noexcept(is_nothrow_invocable_v<F, **designator_list_t<Designators…>**, Args...>);

template<class R, **class… Designators**, class... ArgTypes> class function<R(ArgTypes… **fwdarg(Designators)**)>;

template<class R, **class… Designators**, class... ArgTypes>
void swap(function<R(ArgTypes… fwdarg(Designators))>&, function<R(ArgTypes… **fwdarg(Designators)**)>&) noexcept;

template<class R, **class… Designators**, class... ArgTypes>
bool operator==(const function<R(ArgTypes… **fwdarg(Designators)**)>&, nullptr_t) noexcept;

9. References

[Hardgrave 1976]
Date: 1976-05-??
Title: Positional Versus Keyword Parameter Communication In Programming Languages
Author: W. T. Hardgrave, NASA Langley Research Center

[Hartinger 1991]
Date: 1991-??-??
Doc.No.: WG21 / N0060
Project: Programming Language - C++ (WG21)
Title: Keyword Parameters in C++
Authors: Hartinger, Schmidt, Unruh.
Link:
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1991/WG21%201991/X3J16_91-0127%20WG21_N0060.pdf

[N0088R0 Eckel 1992]
Date: 1992-??-??
Doc.No.: WG21 / N008R0
Project: Programming Language - C++ (WG21)
Title: Analysis of Keyword Arguments (R0)
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1992/WG21%201992/X3J16_92-0010%20WG21_N0088.pdf

[N0088R1 Eckel 1992]
Date 1992-??-??
Doc.No.: WG21 / N008R1
http://www.open-std.org/jtc1/sc22/wg21/docs/papers/1992/WG21%201992/X3J16_92-0010R1%20WG21_N0088R1.pdf

[Gibbons 1992]
Date: 1992-??-??
Doc.No.: WG21 / N0131
Project: Programming Language - C++ (WG21)
Title: Second Analysis of Keyword Arguments (C++)
Authors: Bill Gibbons, 1992
Link:
http://www.open-std.org/JTC1/sc22/wg21/docs/papers/1992/WG21%201992/X3J16_92-0054%20WG21_N0131.pdf

[Stroustrup 1994]
Date: 1994-??-??
Title: The Design and Evolution of C++, Section 6.5.1 Keyword Arguments
Author: Bjarne Stroustrup

[Ballo 2014]
Date: 2014-10-07
Doc.No.: N4172
Project: Programming Language - C++ (WG21)
Title: Named Arguments
Authors: Ehsan Akhgari <ehsan@mozilla.com>, Botond Ballo <botond@mozilla.com>
Link: http://wg21.link/n4172

[EWG 2014]
Date: 2014-??-??
Project: Programming Language - C++ (WG21)
Subgroup: Evolution (EWG)
Title: Meeting notes on N4172 [Ballo 2014]
Link:
https://wiki.edg.com/bin/view/Wg21urbana-champaign/FunctionFacilities#N4172_Named_Function_Arguments

[Arena 2014]
Date: 2014-12-16
Article: Bring named parameters in modern C++
Author: Marco Arena
Link: https://marcoarena.wordpress.com/2014/12/16/bring-named-parameters-in-modern-cpp/

[Naumann 2018]
Date: 2018-05-07
Doc.No.: P0671
Project: Programming Language - C++ (WG21)
Title: Self-explanatory Function Arguments
Author: Axel Naumann <axel@cern.ch>
Link: http://wg21.link/p0671

[Brown 2018]
Date: 2018-10-08
Doc.No.: P1229
Project: Programming Language - C++ (WG21)
Authors: Jorg Brown <jorg.brown@gmail.com>
Title: Labelled Parameters
Link: http://wg21.link/p1229