Abbreviated Parameters

Document #:	xxx
Date:	2021-19-06
Project:	Programming Language C++ SG17

Reply-to:	Mihail Naydenov <mihailnajdenov@gmail.com>

1 Abstract

This paper suggests alternative way of declaring parameter lists, one that let us omit parameter types.

2 Background

Lambda expressions are often used in a specific context, which can be used to infer some of their required elements. Because of this, there is a strong desire to have the shortest possible form of such expressions. This resulted in few proposals, like for example **Abbreviated Lambdas (P0573R2)**[1]. P0573R2 did not pass for few reasons,[2] here is a summary:

— **Differing semantics with regular lambdas.** Means that the same function body will return different types, depending if it as an abbreviated lambda or normal one.

— **Arbitrary lookahead parsing.** Means the parser must skip up to the beginning of the body of the lambda to know if it deals with omitted types or not.

— **Mismatch between the trailing-return-type and the body** As the name suggests, the return type and the body are parsed differently, making the Abbreviated Lambdas might fail to perform as intended.

It is not hard to notice, two of the issue are related to the semantics of the Abbreviated Lambdas **body**, one is related to the **parameters**.
This paper also makes the observation, often it is the params that are contributing to the verbosity of a lambda the most, if we focus on the day-to-day uses and not the "perfect forwarding" scenario:

```
[](const auto& a, const auto& b) { return a < b; }
```

Parameter types dominate the lambda expression in common cases.

> An argument can be made, we could use `auto&&` and get rid of `const`, saving significant number of characters. This is not entirely true, because, `auto&&` will be a `const` reference only if the original object is `const`. *Most* of the times this is not true - the object is not `const`, yet we want to immutably operate on it. By using `auto&&` we would change the meaning of our lambda, no matter how we look at it.

Interestingly, not only we will gain the most if we get rid of the types, sans "forwarding", but it seems, the problems we had in attempt to do so are mostly technical - no easy way to differentiate b/w normal parameter list and one with no types. The **body** issues on the other hand are significantly more involved, while at the same time giving smaller verbosity reduction in day-to-day code.

This paper suggest splitting the issues in **"body issue"** and **"params issue"** and deals only with the latter.

3 Proposal

Introduce a *slightly* different syntax to denote **Abbreviated Parameters**:

```
// lambda:
[]((a, b)) { return a < b; }
```

[1] Abbreviated Lambdas: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0573r2.html
[2] Barry Revzin blog: https://brevzin.github.io/c++/2020/01/15/abbrev-lambdas/

```
// function:
auto less_then((a, b)) { return a < b; }
```

Double parenthesis indicated Abbreviated Parameters.

3.1 Details

If an parameters list starts with double parentheses ((, then all single identifier are *not* types, but variables instead:

```
auto something(string, vector);   //< declaration of function with two parameters of type `string` and `vector`.
                                  // param names are omitted
auto something((string, vector)); //< declaration of function with two parameters of deduced type.
                                  // param names are `string` and `vector` (this proposal)
```

Using multiple identifiers results in exactly the same declarations no matter if single or double parentheses are used:

```
auto something(vector v);   //< same declarations
auto something((vector v)); //<
auto something(const vector& v);   //< same declarations
auto something((const vector& v)); //<
```

Mixing single and multiple identifiers is possible:

```
auto something((string, const vector& v)); //< `string` is of deduced type, `v` is of const vector&
```

In other words, **types are optional**. Parameters of deduced type are **templated parameters**. Functions and lambdas, containing such parameters are templates. The above declaration is equivalent to:

```
template<class T>
auto something(T <qualifiers> string, const vector& v);
```

Function template. `<qualifiers>` *are discussed in* ***Part 2****.*

Mixing with regular template parameters is also possible:

```
template<class T> void func((T a, b));
[]<class T>((T a, b)) {};

// equivalent to
template<class T, class U> void func(T a, U <qualifiers> b);
[]<class T, class U>(T a, U <qualifiers> b) {};
```

As well as omitting identifiers, partially or completely:

```
void func((,));
[]((a,,c,)) {};

// equivalent to
template<class T, class U> void func(T <qualifiers>, U <qualifiers>);
[](auto <qualifiers> a, auto <qualifiers>, auto <qualifiers> c, auto <qualifiers>) {};
```

If there are no identifiers and no commas, the list is empty:

```
void func(());

// equivalent to
void func();
```

If the parameters list does not contain deduced types, or extra commas, the declaration is *not* a template:

```
void func((T a, U b));

// equivalent to
void func(T a, U b);
```

Default arguments can be supplied as usual:

```
void func((a = 12, b = "hello"s));

// equivalent to
template<class T=int, class U=std::string> void func(T <qualifiers> = 12, U <qualifiers> b = "hello"s);
```

The syntax can not be used to declare function pointers or signatures:

```
// using FuncPtr = void(*)((string));                 // error
// using FuncPtr = void(*)((std::string));            // error
// using FuncPtr = void(*)((const std::string&));     // error
```

The main reason to disallow this is to be consistent with current `auto` rules:

```
// using FuncPtr = void(*)(auto); // already an error
```

Although, it could be argued `auto` in `using` declaration can be allowed to mean a template param for the `using` declaration itself:

```
template<class T>
using FuncPtr = void(*)(T);
```

As this is completely out of scope for the current proposal, **double parentheses are allowed only where `auto` placeholder is allowed**.
Similarly, **function declarations inside a block are not allowed** :

```
void f()
{
  [](string){};              //< Lambda: OK
//  auto something(string); //< Function: error
}
```

This should come to no surprise, because function templates in block scope are not possible today. However, even if we allow these, double parenths declarations will *still* be disallowed, because it will be a breaking change: `auto something(string));` is already a valid expression (a constructor call to `something` object, taking `string` variable as an argument).

This covers **Part 1** of the proposal - **allow a 3th way of declaring function templates, one that will let us to omit the type (or its placeholder in the form of `auto`) completely**. **Part 2** will discuss the options we have with regard to the deduced type and its decorations.

3.2 Example

As stated in the beginning, getting rid of the parameters type is one of the best "bang for the buck" in the attempt to lower verbosity:

```
// given
struct fruit {...};
std::vector<std::string> fruits{"apples", "oranges","cherries"};
std::vector<std::vector<fruit>> baskets{...};

std::transform(fruits.begin(), fruits.end(), baskets.begin(), fruits.begin(),
// from
  [](const std::string& fruit, const std::vector<fruit>& basket) { return fruit + ": " + std::to_string(basket.s
// or
  [](const auto& fruit, const auto& basket) { return fruit + ": " + std::to_string(basket.size());
// to (this proposal)
  [](fruit, basket)) { return fruit + ": " + std::to_string(basket.size());
});

// `auto&&` is discussed in Part 2
```

If we take a theoretical "abbreviated body", the gains are much less in terms of reducing verbosity, at least in this example, though I would argue, the example is fairly representative in general:

```
std::transform(fruits.begin(), fruits.end(), baskets.begin(), fruits.begin(),
  [](const auto& fruit, const auto& basket) { return fruit + ": " + std::to_string(basket.size();
});
std::transform(fruits.begin(), fruits.end(), baskets.begin(), fruits.begin(),
  [](const auto& fruit, const auto& basket) => fruit + ": " + std::to_string(basket.size()
);
```

As you can see, *if we ignore the generic, forwarding case*, the gains are almost non-existent - **a constant 7 characters total,** 6 of them from `return`, 1 for the semicolon.
This is not to say the generic case is to be ignored - quite the contrary. *Most* of the benefits in an "abbreviated lambdas" will come from a new, smarter "abbreviated body". An "abbreviated body" will solve more problems then simply limiting verbosity as it will lower the complexity of advanced topics that only look simple on a first glance. This paper is ***not*** against any "abbreviated body" solutions and it does not block or contradict ony of them.
In any case however, "abbreviated parameters" have enough practical benefit for "abbreviating" the lambda expression as a whole to be worth pursuing on their own.

3.3 Related Work

Using duplicated symbols to denote a different entity is not new. In fact, it can be traced back to (at least) `C`:

- We have *multiple* operators that have double versions `||`, `&&`, `++`, `--`, `<<`, `>>`, `==`;
- We have two types of references, `&` and `&&`, expressed via duplication of the symbol;
- We have both lambda capture - `[something]` - and attributes - `[[something]]`;
- We have both `:` and `::`;
- We have `.` and `...`;

I would argue, symbol duplications is at this point a natural way to introduce new meaning without new symbols.

3.3.1 The bigger picture

Although this proposal is focused on the parameters part of the function/lambda declaration, arguing for the benefits of its own, it can also be seen as enabler for the complete Abbreviated Lambda (P0573R2[3]) proposal as well. Recently there is a push to solve the 3th issue that prevented its adoption, the **Mismatch between the trailing-return-type and the body** part, with the introduction of **p2036r1**.[4] It proposes changing the return type parsing to be identical to function body parsing, negating the said issue.
If current and the **p2036r1** proposals are accepted, we will have 2/3 issues solved, which is much, much better place to be!

Here is the place to emphasize, **all** modern languages, have *both* abbreviated params for lambdas and lambdas expression bodies: **D, C#:** `(x, y) => ...` , **Java:** `(x, y) -> ...` , **Rust:** `|x, y| ...`, **Kotlin:** `{x, y -> ...}`, **Swift:** `{x, y in ...}`
These are *non-functional*, *strongly-typed* languages, all agreeing types in this context serve little to no value while greatly increasing the visual clutter. For C++ it is no different.

4 Future direction (not proposed for now)

Interestingly, if the current proposal is applied to `range-for`, *it solves the main issue, which led to the rejection of the* `for(elem : range)` *syntax.*
The problem was that declarations like this can be misleading[5]:

```
int i;
for(i=0; i<size; i++)     //< these are different - reuse-vs-introduce
for(int i=0; i<size; i++) //<
something i;
for(i : container)     //< are these different as well?
for(int i : container) //<
```

Having a syntax that *already* means variable introduction (of deduced type) solves this issue completely:

[3] Abbreviated Lambdas: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0573r2.html
[4] Change scope of lambda trailing-return-type: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2036r1.html
[5] range-for discussion: https://groups.google.com/a/isocpp.org/g/std-proposals/c/nKRCQVvCxD8

```
something i;
for((i : container))   //< obviously an introduction
for(int i : container)
```

*Double parentheses **always** mean variable introduction.*

5 Part 2

This part deals with what the actual deduced type should be.

The often regarded "right solution" for that is to use `auto&&` - it is what **p0573** suggested and was previously proposed for the `for(elem : list)` idea as well.
This proposal argues against it, *though not strongly.* **It is a valid option** and the current proposal, at its core (Part 1), will be equally well served by any type deduction. With that out of the way, let's see pros and cons of other options.

5.1 Downsides of `auto&&`

5.1.1 Mutability

The single biggest downside of `auto&&` is its mutable reference semantics as default. These are not desirable defaults.
Lets return to our fruits example:

```
std::transform(fruits.begin(), fruits.end(), baskets.begin(), fruits.begin(),
  [](auto&& fruit, auto&& basket) { return fruit += ": " + std::to_string(basket.size());
});
```

If the user incidentally assigns to `fruit`, it will break `std::transform` preconditions.
It could be argued, the user can use `fruits.cbegin()` instead of `fruit.begin()`, but this requires *the same amount of attention* as writing `const` for the lambda params. If we talk about safe *default*, `auto&&` is not one.
Besides in some cases **`cbegin()` is not even an option:**

```
std::sort(fruits.begin(), fruits.end(),
    [](auto&& lhs, auto&& rhs) { return lhs.price() < rhs.price(); }
);
```

What if `price()` is not the correct method (or overload!) as it, for example, fetches price from an external source, does some other processing, and is undesirable in this context for performance or other reasons? We created a problem where there was none before. If this is example looks contrived, then why do we have `begin()` and `cbegin()` in the first place?

It gets even more interesting, if we take into account how predominant `const` lambdas are in practice. For example, *from all standard algorithms, **only one** is usable with mutating lambda - `for_each`!* Even if the algorithm itself is mutating, the function object itself is not, instead the return value is used to write to desired destination (either to the same container or not, resulting in either mutation or copy).

An argument can also be made about forwarding scenarios, where non-const ref-ref is the desired type. However, forwarding specifically is excursively a library/wrapper feature. These scenarios are by definition less common then "the general usage" - a "library" is meant for code-reuse, and code-reuse is "*write once*, reuse many times". This is, the day-to-day usage of a language is always more common then writing a library for it.

5.1.2 Hidden reference

Using `auto&&` in the case of `[](a, b){}` (or `for(elem : list)` for that matter) creates non-obvious and arguably unexpected hidden reference.
If we compare current syntax for

- variable: `int i`
- parameter: `void(int i)`
- capture: `[i]{}`

We never create reference, that is not explicitly marked in syntax:

- variable: `int& i`
- parameter: `void(int& i)`
- capture: `[&i]{}`

Given this, **what the expectations will be in this scenario**:

```
[](a, b){ a++; b += a; return b;}
```

Let's be honest, **no one would expect some far-away state is altered**, *unless* he/she is *intimately* familiar with how are deduced types implemented. And it is not so obvious, not obvious at all. For example one might expect "behind the scenes" the declaration be in the form:

```
[](auto a, auto b){ a++; b += a; return b;}
```

This is not unreasonable assumption - after all *this is how capture works today* (mutability aside)! As said, one must have *intimate* knowledge to guess the answer.

Interestingly, or even ironically, the hidden nature of the reference works against the forwarding case as well:

```
[]((a)){ return some_func(std::forward<decltype(a)>(a)); }
```

Are we really doing forwarding here? How can one tell?
Because there is no reference in sight, let alone a `&&` one, the user again must have intimate knowledge and trust, this works as expected - after all 1/2 visual hints that marks forwarding scenario is gone. And the hint that is removed is pretty important - only a specific type of reference is a forwarding one!

5.1.3 Verbose to get `const` back

If we consider immutability is always a good thing, how do we get it back?
We either need full verbosity:

```
[](const auto& a, const auto& b){...}
```

Or, well, partial verbosity, if we extend the possible definitions to allow "naked `const`"

```
[]((const a, const b)){...}
```

Considering, for many types, we can just use non-reference `auto` today, we did not achieve much.

For an **Alternative solution** see **Appendix**.

5.2 Proposed solution

This paper suggests, a **const reference** and/or **const copy** as the best options for the deduced type.
More specifically, **const reference** to be the default, but, *if deemed feasible*, **allow optimizers to use copy instead** for improved performance.

```
std::string s;
std::vector<std::string> v;

std::for_each(s.begin(), s.end(), [](c){ ... }); //< decltype(c) is `const char`
std::for_each(v.begin(), v.end(), [](s){ ... }); //< decltype(s) is `const std::sting&`
```

Because in both cases we are dealing with a `const` arg. the effect of the transformation to copy will be close to invisible to the user. The only observable side-effect will be if the user, or a subroutine, takes and stores the address of the argument, with the assumption, its lifetime is greater then the lambda itself. In that case the address can become invalid or not, depending on the argument type.
How much this is a problem in practice is hard to jude, but this paper argues, it is extremely unlikely, a user will intentionally take the address of a param (or call a function which does so) AND use a signature, which does *not* have reference/pointer symbol on the param:

```
void func((arg)) { persistAddrOf(&arg); ... } //< Unrealistic scenario
```

It is unlikely, the user expects, `arg` storage to outlive the call to `func`.

This goes back to the fact - there are no hidden references in C++. Sure, arg *could* be a reference, but no reason for the user to *expect* that. There is no previous experience to create such expectations!
Also, the implementation can be such that, the user can query and/or assert if the conversion to reference will take place. Something along the lines of `static_assert(sizeof(arg) < TO_CREF_THRESHOLD)`. This will allow 100% safe code, albeit with increased complexity.

Having said that, I do see this feature as potentially controversial, that is why, it is suggested only if feasible - **const reference alone is good enough default**. Besides, if the user wants to have a copy instead, a simple `auto` is still an option.

While on the subject of copying, why aren't the arguments passed by copy, the way they are passed in functions and lambdas already, if there are no cvref decorations?
There are multiple arguments against that.

— Expensive to copy big objects.
— Verbose to get immutability back, even more so to get both reference *and* immutability.
— Not really possible to implement "maybe-reference", because this will change how the function/lambda works in a dangerous way. We can get away having "maybe-reference" exclusively because immutability, which makes the transformation safe, as long as one operates on the param alone, not its address.

The pro-arguments are much weaker:

— No hidden immutability.
— No hidden const reference.
— No hidden change of param address. (In case "maybe-reference" is implemented)

The first argument is weak, because we have a precedent with lambda captures. Captures can be though as "arguments, passed on lambda definition". Not only that, but the immutability of lambdas has *overwillingly positive* reception in the community. In a way, by making the parameters of a lambda immutable, *we increase the overall consistency of the lambda expression as a whole* - now *all* arguments, passed to a lambda, either on definition or call time, are immutable.
The second argument is also weak, because it does not affect how we write code - the reference is there purely for performance reasons, exactly how it is used today.
The issue about param address is already discussed.

> A note about **coroutines**. This paper does not explore this space, but suggests, it is good idea to have different rules for deduction for coroutines. This is because references and coroutines do not match well. It looks like capture by copy to be much better option, however this is a separate discussion and **for now coroutines should *not* support double parenths syntax.** We simply don't have enough experience with coroutines to be sure of the right defaults. Coroutines are significantly enough different, then either lambda or functions, and require a separate evaluation. Extending the current proposal can be done on later date.

The above scenario handles "observation" functions only. If the user wants to modify the original argument passed (or want to ensure it has its address passed to the function), then using both `&` or `&&` are proposed to serve these purposes:

```
auto& incr((&src, delta)) { return src += delta; } //< `src` is of type auto&, `delta` is of type `const auto(&)

[]((&&a)){ return some_func(std::forward<decltype(a)>(a)); } //< `a` is of type auto&&
```

By covering the 3 scenarios above ("observation", "mutation", "forwarding") we solve all the issues listed above.

— Const is the default, matching the majority of cases and being the safest option.
— No hidden reference*. Normal references are introduced as usual (matching the capture syntax completely). Forwarding references are clearly visible, as always. *At no point there is **any** potential confusion or uncertainty what the code does!*
— No hidden mutability, yet easy to introduce when needed, with a well know, minimal syntax.

*The immutable reference is an implementation detail, an optimization, and is not *semantically* relevant.

In should be noted, so far `auto&&` as a default was ever only suggested in a very limited context - in a for-loop and lambdas (or lambda-like, single expression functions). Current proposal is not limited to those type of uses alone and anticipates typeless params potentially in normal functions as well. This changes the requirements a bit, as being able to express all common parameter usages ("observation", "reference", "forwarding") is of much greater value. This is, while the niche use case of for-loop or lambda can be handled by one "good enough" type, a more universal support of deduced params require greater flexibility and expressiveness.
For example, consider helper functions, local to a cpp file, or a private function in class. In these scenarios it is quite possible, the user might not use concepts to define such functions and opt for the "quick-and-dirty" approach as the functions are not really part of an API and the extra effort to "do it right" might not be worth it. Or may be this is a changing code, in a prototyping stage - the user is unsure what the requirements are. In these scenarios it will be highly unlikely, the "forwarding reference" to be the right type (even less so for all arguments!), and is quite likely, the user will want *some* control over the parameters, at least in terms of mutability:

```
namespace {
  void assignPedalCrankFromRotorSpeed((&pedal, rotor)) { pedal = rotor / CHAIN_FACTOR; }
```

```
  ...
}
```

A local helper function

5.2.1 Pro and Contra `auto&&`

As said in the beginning, `auto&&` is reasonable solution as well. Its main strength is its simplicity both in terms of definition and implementation, as well as the fact "it just works" in code. There is a charm to that.
The "just works" part is considered especially relevant in cases where a mutating param must bind to a proxy object:

```
std::vector<int>  vi(10);
std::vector<bool> vb(10);
std::for_each(vi.begin(), vi.end(), [](auto& v) { v = 1; });    //< works
std::for_each(vb.begin(), vb.end(), [](auto& v) { v = true; }); //< fails: MUST be auto&&
```

There is no denying, "it will be nice" to be able to write a minimal expression, which will work in both cases, even more so, considering the cases where proxy objects are used are expected to increase somewhat with `ranges`.
However, this is only true from *modifiable* references, *normal observation works with no added verbosity:*

```
std::vector<int>  vi(10);
std::vector<bool> vb(10);
std::for_each(vi.begin(), vi.end(), [](v)) { if(v == 1) ... ; });    //< works with current proposal as well
std::for_each(vb.begin(), vb.end(), [](v)) { if(v == true) ...; });  //<
```

This leaves us back to the fundamental question: **Is it worth trading const?** Consider:

```
std::vector<int>  vi(10);
std::vector<bool> vb(10);
std::for_each(vi.begin(), vi.end(), [](v)) { if(v = 1) ... ; });    //< "just works"?
std::for_each(vb.begin(), vb.end(), [](v)) { if(v = true) ...; });  //<
```

Do we want to drop safer code for more "easy" one? Do we want to **not** guard against the above code, so that we can have "nice, universal syntax" *in the limited context of mutation* ***alone***? It is an honest question, and although this paper argues for const, the opposite still has merit.

Related to that also is the question, **do we want less clear code if favor of marginally less verbose.**
Is this **"better":**

```
[](a)){ return some_func(std::forward<decltype(a)>(a)); }
```

then

```
[](&&a)){ return some_func(std::forward<decltype(a)>(a)); }
```

— Is it "better" to forward an argument that is not decorated with `&&`?

— Do the `&&` "tip the scale" and make the declaration "verbose"?

This paper expresses doubt. Where `auto` gives no additional information and can be cut out, `&&` *does* convey information!
Of course "people will learn", but it is not that simple. Where now (with `&&`) one can instantly see what kind of function he/she is dealing with (is it forwarding or not), when there is no such indication, all bets are off until the entire body is read:

```
auto function((a, b, c)) {
  ...
  // 10 lines later
  something(a);
  ...
  // another 20 lines
  other_thing(std::forward<B>(b));
  ...
  // yet another 30 lines
  return do_it(std::forward<C>(c));
}
```

What are the chances, later a colleague comes in and:

```
  ...
  // another 20 lines
  other_thing(std::forward<B>(b));
  ...
  // yet another 30 lines
  c.omn(b); //< ok?
  ...
  return do_it(std::forward<C>(c));
}
```

What if `b` is moved by `forward`? Did the colleague noticed the `forward` at all, somewhere in the middle of the function?

Mistakes like this are best handled by static analysis for sure, but having more expressive parameters does not hurt either:

```
auto function((a, &&b, &&c)) { //< obviously not just observing
  ...
}
```

The && is a clear hint, there might be forward at some point!

Less is not always more.

5.2.2 Summary

default to `auto&&`

+ Simple implementation
+ Universal use in all cases
+ Minimal verbosity

- Less safe (or more verbose)
- Less clear
- Learning curve for confident use

default to `const auto(&)`, allow `&` and `&&`

+ More safe
+ More clear
+ More control
+ Minimal-to-No learning curve, consistent with current practices overall
+ (Better performance if conversion to `const auto` is feasible)

- More involved implementation
- Marginally more verbose *in some cases*
- "Pedantic"?
- (Even more involved with `const auto` conversion)

The above summary shows, no solution is really "better" - there are trade-offs in both cases.
This paper only hopes, we make the most informed decision when choosing one trade-off over another.

6 Appendix

6.1 Alternative solution

The `auto&&` vs `const auto&` debade can change considerably if the core of the proposal (Part 1) is extended to allow specifying mutability for all arguments:

```
void function((a, b)const); //< All arguments are immutable
```

This shifts the optics about using `auto&&` as it becomes easier to gain immutability (as well as potential "maybe-reference" implementation).
Having such option does not contradict the const reference default as proposed, but makes many aspects of it redundant, like for example allowing `&` and `&&` syntax:

```
void function((&a, &b)const); //< redundant with `const auto&` default
```

Some of the arguments pro explicit references still stand, but are greatly dampened, that is way this is more of an "Alternative solution", not an "extension" - const ref default + parenth-const is *not* propsed.
Parenth-const creates some interesting possibilities, unrelated to abbreviated parameters, like allowing us to have all-const parameters in normal functions:

```
void function((std::string& s, std::vector<int>>& v, int i) const ); //< all arguments are immutable
//And/Or may be const in front
void function(const(std::string& s, std::vector<int>>& v, int i)); //< all arguments are immutable
```

As you can see, no abbreviated parameters in sight - we only take advantage of having to type just one `const` for all params. Of course, the value of this feature greatly depends on the number of parameters (and if we have a mix of const and non-const params), but is interesting nevertheless.

Arguably, the most interesting results are when this is applied to the theoretical `for-range` extension:

```
// `auto&&` is the default + parenth-const + range-for extension

for((e : list)const)  //< some verbosity for observation, but clear and safer

// equivalent to
for(const auto&& e : std::as_const(list)) //< and immutable container!

for((e : list)) //<  minimal-verbosity mutation and universal

// equivalent to
for(auto&& e : list)
```

As shown, this could solve the age-old problem where the list itself is not immutable, leading to undesired side-effects in some situations like when used with ref-counted containers.

Should be noted, when `e` is by default deduced to `const auto&` we can simply re-define the expression to have immutable container, if there is no reference on the element variable. In other words, we can get this feature (a const container) **for free**, because of the immutable default. The downside is that we lose minimal-verbosity mutation and we have to use `&&` to be universal.

```
// `const auto&` is default _alone_ + range-for extension

for((e : list)) //< minimal-verbosity observation

// equivalent to
for(const auto& e : std::as_const(list)) //< and immutable container!

for((&&e : list)) //< to be universal, we need &&

// equivalent to
for(auto&& e : list)
```

6.1.1 Why this + `auto&&` is not the primary proposal?

- More verbose for the common, observation case.
- Hidden mutable reference is still the default!
- Introduces more new rules - not exactly a "natural extension" of preceding art.

~ Interesting, but questionable value outside abbreviated parameters. My be with the exception of `range-for`?

+ Allows all mutating cases to be at minimal-verbosity possible.
+ Simpler, no need to introduce both `&` and `&&`.

Unsurprisingly the value of the alternative solution relies once again on how we much we value immutability and what price we want to pay for it.
Do we want default const and can live with "references noise" when modifying or want clear code when modifying and have const as an explicit opt-in, with the usual verbosity that comes with it.