Expression Function Body

Document #:	xxx
Date:	2021-27-06
Project:	Programming Language C++ SG17

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

1 Abstract

This paper suggests a way of defining single-expression function bodies, that is aimed at solving one of the issues, which prevented standardizing the **Abbreviated Lambdas**[1] proposal.

2 Background

2.1 The importance of expression functions

Both library and everyday code often require forwarding one function call to another:

```
// 1. Calling a member
std::all_of(vs.begin(), vs.end(), [](const auto& val) { return val.check(); });

// 2. Binding an object
std::all_of(vs.begin(), vs.end(), [id](const auto& val) { return val.id == id; });

// 3. Passing a lazy argument
auto get_brush_or(painter, []{ return Brush(Color::red); });

// 4. Creating an overloaded set
auto func(const std::string&);
auto func(int) noexcept;
inline constexpr auto func_o = [](const auto& val) { return func(val); };

std::transform(strings.begin(), strings.end(), strings.begin(), func_o);
std::transform(ints.begin(), ints.end(), ints.begin(), func_o);

// 5. Creating an overload
auto pixel_at(image& image, int x, int y) {
  return pixel_at(image, point{x, y});
}

// 6. Creating a simplification wrappers
auto load_icon(const std::string& str) {
  return load_resource(resources, str, "icn", use_cache);
}

auto totalDistance() {
  return std::accumulate(distSinceReset.begin(), distSinceReset.end(), distAtReset);
}

// 7. Creating "make" functions
template<class T, class... Args>
```

[1] Abbreviated Lambdas: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0573r2.html

```
auto make_the_thing(Args&&... args) {
  return the_thing<T, something_t<T>>(std::forward<Args>(args)...);
}

// 8. Creating operator implementations
auto operator==(const A& a, const B& b) {
  return a.id == b.key();
}

// 9. Creating functions delegating to other functions
class Person
{
public:
  auto name() const { return id.name(); }
  void setName(const std::string& val) { id.setName(val); }
  ...
private:
  Identity id;
  ...
};

// 10. Creating generic forwarding objects
auto repack(auto a) {
  auto b = ...;
  return [a,b]<class... T>(T&&... args) mutable { return a.eval(b, std::forward<T>args...); }
}
```

As you can see, the list is fairly long, and probably incomplete, yet none of the examples are obscure, quite the contrary, this is code that exists in literally very codebase.

Also note, normal functions do forwarding not less often the lambdas.

All of these examples however are incorrect and most of them are suboptimal. They are incorrect because they override the exceptions specifiers in all calls and are suboptimal when concepts/SFINAE checks are required. For details see the original P0573R2[2] proposal.

These two problems are fundamental and although they could be solved via extreme verbosity and/or macros, this is not a practical solution in most cases. At this point, to go to the effort, "to do it right", is something only library writers will do (and suffer the pain).

We can do a better job. This was the main motivation behind P0573R2[3], and this is the main motivation of the current paper as well - ***allow correct and optimal code by default***. It is not just about saving few characters.

Interestingly both of the issues seem to get more pronounced over time. Modern concept-based code brings "SFINAE to the masses" - now *anyone* can write a function, overloaded by a type constrain:

```
void something(std::invocable<int> auto f);
void something(std::invocable<std::string> auto f);
```

Yet, calling this with our current "no-brainer" lambda will fail:

```
something([](auto arg){ return arg/2; })
```

This is just one example, but we can expect, the amount of code requiring concepts/SFINAE checks to rise, often *even without the author of the original code realizing it completely* - it comes naturally. To have things "just work" is now of greater value then ever, as checks like this are no longer in "expert", "template magic" code, they will be everywhere.

Arguably, with the exception specifiers, things can get even more interesting with the push for the new "value-based" exceptions[4]:

```
auto func(const std::string&) throws; //< updated to throw statically
auto func(int);
```

[2] Abbreviated Lambdas: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0573r2.html
[3] Abbreviated Lambdas: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0573r2.html
[4] Zero-overhead deterministic exceptions: Throwing values: http://open-std.org/JTC1/SC22/WG21/docs/papers/2018/p0709r0.pdf

```
...
std::transform(vs.begin(), vs.end(), vs.begin(), [](const auto& val) { return func(val); }); //< What will happe
```

In the above code, the original functions switched to static expression. What happens to the lambda? According to the current proposal - it will still use dynamic exceptions, transforming the static to a dynamic one. **Definitely** not what the user had hoped for!

> Another topic to consider are "niebloids"/CPO/CPF. We can speculate, these will have an elaborate implementation somewhere and the CPO will only delegate/forward to it. For CPO both exception specification and especially the concept checks are essential.

2.2 Abbreviated Lambdas

Abbreviated Lambdas[5] is a proposal which aimed to solve the above issues. It was rejected for the following reasons:[6]

— **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.

The last issue will be addressed by **p2036r1**[7].
The second issue will be addressed by a separate **Abbreviated Parameters** proposal.

The current proposal attempts to solve the first issue: **Differing semantics with regular lambdas.**

2.3 What was the issue again?

Abbreviated Lambdas were defined as follows:

```
[]() noexcept(noexcept(_expression_)) -> decltype((_expression_)) { return _expression_; };
```

This means that reference-semantics are the default:

```
// given
int i;
auto l = [](int* p) noexcept(noexcept(*p)) -> decltype((*p)) { return *p; };

// decltype(l(&i)) is int&

// where by default

auto l2 = [](int* p) { return *p; }
auto func(int*) { return *p; }

// decltype(l2(&i)) and decltype(func(&i)) are int
```

Effectively, if the user want to use an abbreviated form, `[](int* p) => *p;` in place of `[](int* p) { return *p; }`, he/she will get a different result.
Sometimes the results can unexpectedly different:

```
[](auto val) => val;
```

The above will return reference to local. Probably not great and not what the user expects. Granted, any compiler on the planet will warn when that happens. Of course returning a reference is often (arguably more often) the desired behavior, including in the pointer example above:

[5] Abbreviated Lambdas: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0573r2.html
[6] Barry Revzin blog: https://brevzin.github.io/c++/2020/01/15/abbrev-lambdas/
[7] Change scope of lambda trailing-return-type: http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p2036r1.html

```
int i;
auto lp = [](int* p) => *p;

int* p = &i;

lp(&i) = 10;
*p += 10;

// i == 20
```

The lambda behaves "as-if" we used a pointer directly.

3 Proposal

Current proposal presents two variants for mitigating the above issue. Mitigating because there is no problem to fix - different semantics are good for different situations. The two alternative are split on what should be the default in minimal expression form.

— **Variant One** will argue, the minimal expression should have reference semantics, but a non-minimal expression will have value-semantics. *This is the main proposal.*
— **Variant Two** will present an option where minimal expression yields value semantics and reference semantics are an "opt-in".

3.1 Variant One (proposed)

This variant envisions "in-between" expression, where the user will get all the benefits of the function body being and expression (exception specifiers, concept/SFINAE friendliness) while still having the value-semantics like normal functions.
This "in-between" expression is achieved simply by removing the curly brackets around an normal, single expression function body with deduced return type:

```
// From
auto pixel_at(image& image, int x, int y) {
  return pixel_at(image, point{x, y});
}
// To (this proposal)
auto pixel_at(image& image, int x, int y) //< no curly braces
  return pixel_at(image, point{x, y});
```

The above will be equivalent to:

```
auto pixel_at(image& image, int x, int y)
  noexcept(noexcept(std::decay_t<decltype(pixel_at(image, point{x, y}))>(pixel_at(image, point{x, y}))))
  -> decltype((std::decay_t<decltype(pixel_at(image, point{x, y}))>(pixel_at(image, point{x, y}))))
    { return pixel_at(image, point{x, y}); }
```

In other words, the behavior regarding the return type is still the same, but with added benefits of exception correctness and concept/SFINAE friendliness.
For example, if the original `pixel_at` returned a reference to the pixel, moving to expression form will not change the behavior and the overload will still return a value. Because overloads returning different types in this way is not realistic, let's change the example a bit:

```
// given
const pixel& pixel_ref_at(const image& image, point p) noexcept;
// From
auto pixel_at(const image& image, int x, int y) {
  return pixel_ref_at(image, point{x, y});
}
// To (this proposal)
auto pixel_at(const image& image, int x, int y)
  return pixel_ref_at(image, point{x, y});
```

After transformation, the new code still returns a pixel by value, but is now also correctly decorated as `noexcept` (if `pixel` is `noexcept` copyable as well).
Preserving the `return` acts as an indicator, one is still in the "function land", where we *call* and *return*:

```
auto l = [](auto val) return val;

// _return_ by value
// decltype(l(1)) is int
```

The above clearly express the notion of passing the value along to whoever called the lambda. The argument is taken, then a value is *returned* back:

```
int i;
auto l = [](auto* p) return *p;

// _return_ the result of dereference
// decltype(l(&i)) is int
```

3.1.1 complete expression function

When we want full "expression functions", we go one step further, away from normal functions, by dropping the `return` as well:

```
auto l = [](auto* p) *p;

// equivalent of
auto l = [](auto* p) noexcept(noexcept(*p)) -> decltype((*p)) { return *p; };
```

The result is "as-if" the dereference was done in the scope that uses the lambda:

```
int i;
auto l = [](auto* p) *p;
int* p = &i;

// decltype(l(&i)) == decltype(*p)
```

This behavior is as **Abbreviated Lambdas** originally proposed.

3.1.2 Motivation

The goal of the proposed solution is twofold:

— Create a "layered" approach, where one has gradual transition b/w ordinary functions and "expression functions".
— Represent the "pure expression" form as closely as possible to "just an expression" with as less syntactical "hints" as possible.

The first goal is achieved by preserving the `return` keyword like a normal function. The `return` serves as a "safety net" against possible surprizes as it *looks* familiar and it makes the expression *act* familiar. In a way, `return` gets you *back* to normal function behavior, where an effort is needed to actually return a reference.

> It is worth stressing out, the situations where `return` is *actually needed* are rare. Vast majority of cases both expressions will behave the same and often even return the same thing. `return` will only be needed *to prevent* returning a reference, not so much *enabling* returning a value - `[](auto& i) i+1;` still returns a value.

The second goal's motivation is to get to the actual expression as close as possible, distancing ourselves from the function metaphor where the body "returns" a value. This is the reason why the `=>` is not used - it is ultimately a different spelling for "a return", and also an evolution of `->`. Where `->` indicates "return type", `=>` indicates "return the expression". The strong notion of returning something is what we try to avoid here. We want a new construct, distant from normal functions and expectations how they work. Take the most simple expression an example:

```
1;
^~~~ expression
```

We want to *lift* that into a construct that will be (re)used at later time. Luckily, we already have distinct lambda introducing syntax. We can use it here without additional syntactical burden (in most cases anyway):

```
auto l = [] 1;
         ~~^~~ "reusable expression"
auto b = l();
         ~~^~~ "reuse"

// more examples
[] ::value;
[] func();
[s] s.member();
```

This paper argues, the above expressions are significantly different then a function to command different understanding. To the very least, it is much less likely, someone will be surprized, that this works:

```
auto l = [] ::value;

l() = 10;

// ::value == 10
```

or this

```
int i = 2;
auto l = [](int& i) i+=2;

l(12)/2;

// i == 2;
```

Once reference semantics mindset is establish, people will not confused or have the wrong expectations. Users will know from experience, the expression *always* yields a reference, *if* there is no temporally created (which is clearly visible when it happens). And even if they get it wrong, like for example trying to pass-along an object by value, the compiler will most certainly issue a hint *"Looks like, you are returning a reference to a local variable. If you want to return a value use return before the expression"*:

```
[](auto a) a;
       ~~~^~~~ add return
```

Of course, there will be cases where the compiler will not be able to "see through", but how much this would be a problem is impossible to predict. One thing that is works in our favor is the fact, this is a single expression. There are not many sources of local variables that could be returned as a reference. These either have to be the arguments or a temporary, created by a function call chaining in the expression itself.

The first is somewhat questionable in practice, because simple arguments tend to be used to create new values as part of the expression - `[](int a) a + 1;` - and complex arguments are in general taken by reference - `[](const Person& p) p.nameAsRef();`. A complex argument taken by value can be used as an optimization technique, but it is rather unlikely to happen on single expression functions. Even if it happens, it will almost certainly be used to create a temporary: `[](Rect r) r.width * r.height;`

The second source for dangling references, function call chaining creating temporaries, is already dangerous and something most programmers are aware of, besides, returning by value only helps to an extent, it's not magic bullet.

More importantly however, returning a reference to temporary is not *necessarily* fatal. Chances are, the reference will be used before the end of the expression that created it:

```
std::partition(begin, end, [](Person p) p.id);
std::transform(begin, end, dest, [](Person p) p.nameAsRef());
...
auto get_person(Person=p{}) p;
const auto person = get_person();
```

Returning reference to local is not always a problem.

In these, and many other situation returning a reference to local is OK. This is because the expressions are "assignment expressions" which, in the end, will find their way either into a value or into a condition expression, both of which are fine in general. Only "clever" code like assignments to a const reference or a `auto&&` is really affected, but this is the exception, not the rule and also can be mitigated by *not* removing the `return` from the expression, if this is an old code that needs to continue to work.

3.1.3 A minor detail

The proposed minimal expression will not be practical in all case. In particular, it might clash with possible future qualifiers:

```
auto Class::func() async;
[](int a) coro;
```

In the above hypothetical examples we will write ourself into a corner by allowing the minimal syntax - all future qualifiers will compete with existing expressions. To counter this, it is proposed to terminate any qualifiers by `::`

```
auto Class::member() const: async;
[](int a) mutable: coro;
```

This way we can introduce new keywords without the danger of clashing with existing expression bodies.
The colon is not needed if `return` is used:

```
auto Class::member() const return async;
[](int a) mutable return coro;
```

Colon not needed if we use ***`return`****.*

The `:` is proposed as it has minimal visual and syntactical clutter. It has long history of being a separator (labels, case labels, parent classes, parent/member constructor calls) and it also does not have any of the "return" association of `=>` from the original paper. If we were to use `=>` here instead, it will be really confusing why sometimes it is not used and how it interacts with `return`. The `:` on the other hand can be presented as just a separator that is not always needed.

> Using `=` was shortly considered, but it is not an option, because it is already used in defaulted, deleted and pure virtual functions.

3.2 Examples

Current implementation

```
class QPointF
{
  ...
  real& rx() { return xp; }
  real& ry() { return yp; }
   real x() const { return xp; }
   real y() const { return yp; }

  friend auto operator+(const QPointF &p1, const QPointF &p2) {
    return QPointF(p1.xp+p2.xp, p1.yp+p2.yp);
  }

private:
  real xp;
  real yp;
};
```

Same return, a bit less typing, ***correct noexcept***

```
class QPointF
{
  ...
  auto rx() xp;
  auto ry() yp;
  auto x() const return xp;
  auto y() const return yp;

  friend auto operator+(const QPointF &p1, const QPointF &p2)
    QPointF(p1.xp+p2.xp, p1.yp+p2.yp);

private:
```

```
  real xp;
  real yp;
};
```

Current implementation (expert)

```
template< class C >
constexpr
auto cbegin( const C& c ) noexcept(noexcept(std::begin(c)))
    -> decltype(std::begin(c)) { return std::begin(c); }
```

Correct by default (noob friendly)

```
template< class C >
constexpr
auto cbegin( const C& c ) std::begin(c);
```

3.3 The Bigger picture

The importance of minimal syntax comes into play when we view it as both alternative to the "overloading set" feature, proposed *multiple* times recently ([8],[9],[10]), as well as a potential stepping stone to the so called "hyper-abbreviated" lambdas[11]:

```
auto func(const std::string&);
auto func(const QString&);
auto func(int) noexcept;
inline constexpr auto func_o = [](const auto& val) func(val);

std::transform(sv.begin(), sv.end(), sv.begin(), func_o);
std::transform(qv.begin(), qv.end(), qv.begin(), func_o);
std::transform(iv.begin(), iv.end(), iv.begin(), func_o);
```

Viable "overloading set" alternative.

Another example, modified from p0119, with the **Abbreviated Parameters** proposal.

```
// Modified from p0119, with
template<typename T>
T twice(T x)
{
  return x * 2;
}

template<typename It>
void f(It first, It second)
{
  std::transform(first, second, first, [](a)) twice(a)); //< Close enough to "lifting"?
}
```

Inline "overloading set" creation.

Going further into some theoretical "hyper-abbreviated" expression is natural:

```
template<typename It>
void f(It first, It second)
{
  std::transform(first, second, first, [] twice(1:)); //< `1:` a "placeholder literal"?
}
```

Here is the place to note, in the minimal form, the space after either `[]` or after `:` **is required**:

[8] Lifting overload sets into function objects (2013) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3617.htm
[9] Overload sets as function arguments (2016) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2016/p0119r2.pdf
[10] Lifting overload sets into objects (2017) http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2017/p0834r0.htm
[11] Now I Am Become Perl (blog):https://vector-of-bool.github.io/2018/10/31/become-perl.html

```
// auto l1 = []something;  //< wrong, space needed after `[]`
// auto l2 = [=]mutable:something.doIt(); //< wrong, space needed after `:`
```

The reason for this to not consume syntax that might be used for other purposes. The `[]` as a prefix was already proposed for both overloading set lifting *and*, for the completely unrelated, language-based support for `std::get<index>(structure)`, the `[index]structure` syntax.
Expecting some use of `:` as a prefix is also reasonable. Currently it is only used for module fragments. Other than that, if the space is not required we will end up with needlessly odd code like this `<modifier>:::value_in_global_sclope.`

Last but not least, minimal syntax is also (extremely) important for lazy arguments usage. From **p2218**[12]:

```
optional<vector<string>> opt = ~~~;
...
// From
auto v3 = opt.value_or_else([] { return vector{"Hello"s , "World"s}; });

// To (this proposal)
auto v3 = opt.value_or_else([] vector{"Hello"s , "World"s});
```

As you can see, minimal syntax get us 99% of "real" lazy arguments.

> Here again it is worth repeating, it is not *just* about the syntax. The exception specification propagation is equally if not more important. A lazy argument *must* have the same requirements as the value it constructs. In the above example, surely both `vector` and `string` throw, but we can image something like `[] Brush(Color::red)`, that might as well have `noexcept` construction.

3.4 Alternatives

Variant One, where the reference semantics are the default, could have two alternatives.

3.4.1 Alternative 1

Use `=>` instead of `return` for the return-by-value option.

Main proposal

```
class QPointF
{
  ...
  auto rx() xp;
  auto ry() yp;
  auto x() const return xp; //< by value
  auto y() const return yp; //

  ...
};
```

Alternative 1

```
class QPointF
{
  ...
  auto rx() xp;
  auto ry() yp;
  auto x() const => xp; //< by value
  auto y() const => yp; //

  ...
};
```

[12]More flexible optional::value_or():http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2020/p2218r0.pdf

We get few characters back, but lose the natural transition from regular functions. It is also questionable if people will be able to remember when to use (or not use) `=>`, where using `return` does have the whole "going back" to a normal function association. This is the reason why this is not the first choice.

3.4.2 Alternative 2

The other alternative is to have just one option - return by reference - and use constructs like `auto(x)`[13] to get a value out of the expression.

Main proposal

```
class QPointF
{
  ...
  auto rx() xp;
  auto ry() yp;
  auto x() const return xp; //< by value
  auto y() const return yp; //

  ...
};
```

Alternative 1

```
class QPointF
{
  ...
  auto rx() xp;
  auto ry() yp;
  auto x() const auto(xp); //< by value
  auto y() const auto(yp); //

  ...
};
```

The benefit of this option that is "simple", as there is only one syntax and getting a value is somewhat "natural", in the sense we don't need "special syntax", but reuse constructs already in the language instead (assuming `auto(x)` gets approved).
There are drawbacks however. The main issue is, this is ultimately a crutch, a "patch up" for the expression that does not behave as desired. It is not exactly natural to get from normal functions to this. In a way we would have two extremes - value (or verbose) normal functions and reference only expressions + "a patch" to get back the old behavior. Using `return` does create a more seamless transition b/w the two.

3.5 Variant Two (Alternative)

This variant flips the defaults, giving precedence of "by value" to be used at minimum syntax:

```
int i;
auto l = [](int* p) *p;

// decltype(l(&i)) is int

// equivalent of
auto l = [](auto* p) noexcept(noexcept(std::decay_t<decltype(*p)>(*p)))
-> decltype(std::decay_t<decltype(*p)>(*p)) { return *p; };
```

Minimal syntax returns a value.

If the user wants reference semantics, the expression must be enclosed with parentheses:

```
int i;
auto l = [](int* p) (*p);
```

[13] auto(x): decay-copy in the language:http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2021/p0849r7.html

```
// decltype(l(&i)) is int&

// equivalent of
auto l = [](auto* p) noexcept(noexcept(*p))
-> decltype((*p)) { return *p; };
```

Using () enables return by reference.

3.5.1 Motivation

Using parentheses to get a reference is already established practice in `return` and `decltype` expressions, used with an identifier or member access:

```
auto l = [](int i) -> decltype(auto) { return (i); };  //< already returns a reference

struct Point { int x; int y; };
auto l2 = [](const Point& p) -> decltype(auto) { return (p.x); };  //< already returns a reference

int i;
decltype((i)) p = i; //< p is a reference
```

Established use of () to get reference for an expression.

Proposed is to have parentheses around the expression behave the same for all expressions:

```
[](Person p) p.id;            //< returns int
[](Person p) (p.id);          //< returns int& (as currently)

[](Person p) p.nameAsRef();   //< returns string
[](Person p) (p.nameAsRef()); //< returns string& (proposed)

// returns a value (or void)
template<class F, class... Args>
auto passthrough(T&& f, Args&&... args) std::invoke(std::forward<F>(f), std::forward<Args>(args)...);

// returns whatever std::invoke returns
template<class F, class... Args>
auto passthrough(T&& f, Args&&... args) (std::invoke(std::forward<F>(f), std::forward<Args>(args)...));
```

Needless to say, in all cases the correct `noexcept` and concept-friendliness are added where appropriate.

Similar to **Variant One**, the minimal syntax will require `:` as separator in some cases, but not when parenths are used:

```
[object]mutable: object.func();  //< by value, minimal syntax, separator after specifier needed
[object]mutable (object.func()); //< by reference, separator not needed
```

3.5.2 Example

Current implementation

```
class QPointF
{
  ...
  real& rx() { return xp; }
  real& ry() { return yp; }
   real x() const { return xp; }
   real y() const { return yp; }

  friend auto operator+(const QPointF &p1, const QPointF &p2) {
    return QPointF(p1.xp+p2.xp, p1.yp+p2.yp);
  }
```

```
private:
  real xp;
  real yp;
};
```

Same return, a bit less typing, *free* `noexcept`

```
class QPointF
{
  ...
  auto rx() (xp);
  auto ry() (yp);
  auto x() const: xp;
  auto y() const: yp;

  friend auto operator+(const QPointF &p1, const QPointF &p2)
    QPointF(p1.xp+p2.xp, p1.yp+p2.yp);

private:
  real xp;
  real yp;
};
```

3.5.3 Why is Variant Two not the main proposal?

There are few reasons for that. Reason one is the speculation, reference being the default will be right, or it will simply not matter - most functions will be used in assignment to value or comparison and/or they will create a temporary either way. That aside, parentheses might seem an obscure way of expressing reference semantics, after all not many people are aware of their usage in `return` and `decltype`. And for people that *are* aware of these uses, it might seem inconsistent to change the rules in such a way, have special handling of parentheses in expressions of this type (and this type alone).

3.6 Conclusion

Presented here were multiple paths to handle *one* of the issues that prevented the original **Abbreviated Lambdas** proposal being accepted. This proposal sees the significant value of having expressions functions bodies not so much for "abbreviating" code, but making it *correct by default*.
The alternatives we have to today are impractical in day-to-day code to the extend, they are not even recommended by experts[14], and/or are insufferable in library code.

Arguably, there are not many features that will improve *both* the code *and* the experience for *both* the library writers *and* regular developers.

---

[14] SO Using member function in std::all_of:https://stackoverflow.com/a/58381525/362515