Constexpr Lambda

Document #: WG21 N4487
Date : 2015-04-28
Revises : None
Project: JTC1.22.32 Programming Language C++
Working Group: Evolution
Reply to: Faisal Vali (faisalv@yahoo.com)
Ville Voutilainen (ville.voutilainen@gmail.com)

Faisal S Vali
Ville Voutilainen
Gabriel Dos Reis

Abstract

This proposal suggests allowing *lambda-expressions* in constant expressions, removing an existing restriction.. The authors propose that certain *lambda-expressions* and operations on certain closure objects be allowed to appear within constant expressions. In doing so, we also propose that a closure type be considered a literal type if the type of each of its data-members is a literal type; and, that if the `constexpr` specifier is omitted within the *lambda-declarator*, that the generated function call operator be `constexpr` if it would satisfy the requirements of a `constexpr` function (similar to the `constexpr` inference that already occurs for implicitly defined constructors and the assignment operator functions).

1 Motivation

In C++14, a *lambda-expression* is prohibited from appearing within a constant expression. Furthermore, operations on closure objects – such as calling the function call operator, the conversion function (to pointer-to-function), or any non-deleted special member functions – are verboten inside constant expressions; either because those functions can not be specified as `constexpr`, or because if a constructor is implicitly defined as `constexpr`, those constructors can not be evaluated as *core constant expressions* (since closure objects are non-literal types). These restrictions on lambda-expressions introduce inconsistencies, surprises and unnecessary restrictions for users of our language.

To paraphrase in code:

```
// The following code is currently ill-formed.
constexpr auto L = [](int i) { return i; };    // NOT OK!

auto L2 = [] { return 0; };
constexpr int I = L2();                        // NOT OK!


// But the functionally synonymous code below is well-formed!
constexpr struct {
  auto operator()(int i) const { return i; }
} L{};

struct {
  constexpr auto operator()() const { return 0; }
} L2{};

constexpr int I = L2();
```

These constraints on *lambda-expressions* and closure objects have not only provoked a national body comment[1] (**FI 8** in the C++14 ballot, that was rejected as too significant a change at that time) but have also compelled library writers such as Louis Dionne – CppCon 2014 presenter and author of the metaprogramming library **Hana**[2] – to petition for `constexpr` lambdas in the next version of C++[3]. He presents the following use case[4]:

> *I would like to (re)open a discussion regarding the allowance of lambdas inside constant expressions. One of my motivations for this feature is the following use case. It is an efficient implementation of std::tuple:*

[1] http://www2.open-std.org/jtc1/sc22/wg21/docs/papers/2014/n3903.html#FI8
[2] https://github.com/ldionne/hana
[3] http://ldionne.com/hana-cppcon-2014/ Slide 36
[4] https://groups.google.com/forum/#!topic/comp.lang.c++.moderated/9Fa2Fzlvlxg

```
template <typename ...Xs>
constexpr auto make_storage(Xs ...xs) {
    auto storage = [=](auto f) { return f(xs...); };
    return storage;
}

template <typename ...Xs>
struct tuple {
    explicit constexpr tuple(Xs ...xs)
        : storage{make_storage(xs...)}
    { }

    decltype(make_storage(std::declval<Xs>()...)) storage;
};

 template <std::size_t n, typename ...T>
 constexpr decltype(auto) get(tuple<T...>& t) {
    return t.storage([](auto&& ...xs) {
        // implementing this efficiently is possible
    });
 }
```

Other authors, such as Paul Fultz[5], have resorted to various machinations to simulate constexpr lambdas. Such issues have manifested often enough, that Richard Smith (implementer of constexpr in clang[6]), in a characteristically illuminating post[7] has asserted:

I'm confident we'll have a good answer for lambdas + constexpr in C++17.

Stroustrup has pointed out that "not everything is best done at compile time"[8]. While we would agree, there seem to be many reasonable uses for constexpr lambdas, especially with algorithms that take predicates or other functors, that can be constant-folded when applied to objects of literal types. It seems that constexpr lambdas aren't inherently wrong, so forcing programmers to resort to more verbose alternative ways to express their designs seems like an unfair imposition.

[5] http://pfultz2.com/blog/2014/09/02/static-lambda/
[6] http://llvm.org/viewvc/llvm-project?view=revision&revision=141561 (and subsequent commits)
[7] https://groups.google.com/a/isocpp.org/d/msg/std-proposals/qcKUf-U7_YU/SRfxv76_ekkJ
[8] http://accu.org/cgi-bin/wg21/message?wg=ext&msg=16688

2 Précis

We propose the following:

1) *lambda-expressions* should be allowed to appear within constant expressions if the initialization of each of its closure-type's data members are allowed within a constant expression:

```
constexpr int AddEleven(int n) {
  // Initialization of the 'data member' for n can
  // occur within a constant expression since 'n' is
  // of literal type.
  return [n] { return n + 11; }();
}
static_assert(AddEleven(5) == 16, "");
```

2) The closure type should be a literal type if the type of each of its data-members is a literal type. This would allow the relevant special member functions to be `constexpr` (if not deleted) and thus evaluatable within constant expressions:

```
constexpr auto add = [] (int n, int m) {
  auto L = [=] { return n; };
  auto R = [=] { return m; };
  return [=] { return L() + R(); };
};
static_assert(add(3, 4)() == 7, "");
```

3) The `constexpr` specifier should be allowed within the *lambda-declarator* to specify the function call operator (or template) as `constexpr`:

```
auto ID = [] (int n) constexpr { return n; };
constexpr int I = ID(3);
```

4) If the `constexpr` specifier is omitted within the *lambda-declarator*, the function call operator (or template) is `constexpr` if it would satisfy the requirements of a `constexpr` function:

```
auto ID = [](int n) { return n; };
constexpr int I = ID(3);
```

5) The conversion function (to pointer-to-function) should, if it exists, be `constexpr`. If the corresponding function call operator is `constexpr`, the conversion function shall return the address of a function that is `constexpr`:

```
auto addOne = [] (int n) {
   return n + 1;
};
constexpr int (*addOneFp)(int) = addOne;
static_assert(addOneFp(3) == addOne(3), "");
```

3 Details and Technicalities

As a caveat, this section discusses technicalities using examples that some readers might consider disturbing and irresponsible. Through these (potentially epileptogenic) examples we hope to facilitate exposition (and not trigger PTSD in those who have suffered through enough uncivilized C++ code). The ensuing analysis is aimed at formalists who concern themselves with the details that harmonize or conflict with the rest of the language's intricacies – other readers are encouraged to skip this section.

3.1 A lambda-expression should be a core-constant-expression if the initialization of its data members are core constant expressions.

We propose that the evaluation of a lambda expression be a *core constant expression* if the initialization of all of its data members (that correspond to each capture) are *core constant expressions:*

```
// The following contrived function definition is valid C++14.
constexpr int eval_lambda(bool eval) {
  // X is a literal type (even though it is a "stranger" type than
  // most closure types) with its default and copy constructor
  // inferred as constexpr.
  struct X {
    constexpr X makeX() { return X{}; }
    union {
     int i = 10;
     double d;
    };
  } x;

  // These initializations are core-constant-expressions in C++14
  auto x1 = x.makeX();
  { auto x = x1; }

  // The initialization of the init-capture x1 and the byvalue
  // capture of 'x' below invoke makeX(), and X's default and copy
  // constructors – which are core constant expressions similar to the
  // code above.
  // Therefore we proppose that the lambda expression below also be a
  // core constant expression.

  // NOTE: Even though the return statement within the lambda
  // expression's compound-statement can not be evaluated as a core
  // constant expression (lvalue-to-rvalue conversion on a volatile),
  // the lambda-expression itself can be evaluated as a
  // core-constant-expression – although invocation of its
  // corresponding closure object's function call  operator can never
  // be a core constant expression.
```

```
  if (eval)
    [x1 = x.makeX(), x] { extern volatile int V;  return V; };
  return 0;
}
static_assert(eval_lambda(false) == 0, ""); // OK in C++14

// Also OK in C++14.
void run_time(bool b) {
   int Z = eval_lambda(b);
}

// NOT OK in C++14, BUT OK under this proposal.
static_assert(eval_lambda(true) == 0, "");
```

Once a *lambda expression* can be a *core constant expression,* they can be evaluated in `constexpr` functions within *constant expressions*. If all their data-member initializations are not guaranteed to be *core constant expressions*, the `constexpr` function has to be invoked with the right arguments for the *lambda-expression* to be a *core constant expression*. This is analogous to certain behavior in C++14. Consider:

```
struct Literal { };
struct NonLiteral : Literal {
  NonLiteral() { };
};

constexpr int eval_lambda2(bool eval_nonliteral) {
  // #1 below is OK in C++14.
  auto Lit = eval_nonliteral ? NonLiteral{} : Literal{};      // #1
  // #2 below should also be OK under this proposal.
  [l = eval_nonliteral ? NonLiteral{} : Literal{}] { };       // #2
  return 0;
}


static_assert(eval_lambda2(false) == 0, ""); // OK in C++14, if #2 is
                                             // commented.
                                             // OK under this proposal
                                             // at #1 & #2.
```

While, at least one of the authors remains unconvinced that the notion of literal types is important to specifying constexpr, that discussion is certainly beyond the scope of this paper (and has been presented before by others[9]).

[9] http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2011/n3308.pdf

3.2 The closure object should be a literal type if the type of each of its data-members is a literal type.

A closure type in C++14 can never be a literal type – even if all its data members are literal types – because it lacks a constexpr constructor that is not a copy or move constructor. If such a closure type was allowed to have an implicitly defined default constructor it would be constexpr, making it a literal type. But, because closure types, by definition, must have their default constructors deleted, the implementation is prohibited from implicitly defining one. Additionally, even though such a closure type has its implicit copy or move constructor be constexpr, that constructor can never be invoked as a *core constant expression* because the enclosing type is not a literal type.

Since we wish to allow constexpr functions to define and return certain closure objects, it is important that their copy and move constructors be invokable as *core constant expressions*. Therefore, we propose that closure types whose data members are literal types be considered a literal type. Under this proposal, the following code would behave uniformly:

```
// In C++14, for the type 'Literal' below the implicitly defined copy
// and move constructor are constexpr. And even though the default
// constructor is deleted, it counts as a constexpr constructor, and
// makes it a literal type.
// This is somewhat analogous to a closure type that has its default
// constructor deleted (such as the type of Lambda below) - but
// admittedly not the exact same.
struct Literal {
  constexpr Literal() = delete;
  Literal(void *) {}
  int operator()(int n) { return n; }
};
Literal Lit{nullptr};

// The type of 'Lambda' could be considered as very similar to the type
// of 'Lit'
auto Lambda = [](int n) { return n; };

template<typename T>
constexpr int foo(T t, int n) {
  T t2{t};
  return n;
}

static_assert(foo(Lit, 5) == 5, "");    // OK in C++14.
static_assert(foo(Lambda, 5) == 5, ""); // NOT OK in C++14.

// OK per our proposal:
constexpr auto make_lambda(int n) {
  return [=] { return n; };
}
constexpr auto L = make_lambda(5);
```

3.3 The `constexpr` specifier should be allowed within the lambda declarator, and if not specified, inferred for the function call operator (or template).

A constexpr function is evaluatable within a constant expression (i.e. at compile time). For a function or function template to be specified as constexpr, it must satisfy some minimal constraints[10] by avoiding: goto; static, thread_local, non-literal and uninitialized variable definitions; asm declarations; try blocks; non-literal types as parameters or as a return type. Admittedly, what constitutes a well-formed constexpr function and what doesn't can surprise some. Consider:

```
// This is a valid C++14 constexpr function, even though it contains
// a potential read from a volatile, a lambda expression and a label!
constexpr int foo1(int n, bool runtime_only = false) {
  int i = n + 10;
  if (runtime_only) {
    volatile int V = 0;
    [&] { i += V; }();
  }
  return i;
UnReferencedLabel:

}


// This is not a valid C++14 constexpr function. It contains an
// uninitialized variable somewhere in the body, even though
// it will never be evaluated.
constexpr int foo2(int n) {
  int i = n + 10;
  if (false) {
    int V;
  }
  return i;
}
```

[10] N4431 7.1.5 [dcl.constexpr]/3 The definition of a constexpr function shall satisfy the following constraints:
— it shall not be virtual (10.3);
— its return type shall be a literal type;
— each of its parameter types shall be a literal type;
— its function-body shall be = delete, = default, or a compound-statement that does not contain
  — an asm-definition,
  — a goto statement,
  — a try-block, or
  — a definition of a variable of non-literal type or of static or thread storage duration or for which no initialization is performed.

```
// This is valid (might involve some serious squinting in the standard)
// in Clang.
constexpr int foo(int n) {
  struct NonLiteral {
       NonLiteral(int) { }
       constexpr int get(int n) { return n; }
  };
  extern NonLiteral NL;
  return NL.get(n);
}
```

While one could claim that some of these restrictions on constexpr could be perceived as confusing and inconsistent, a coherent argument to support that claim is beyond the scope of this paper. What is in scope is that in C++14 for a function to be evaluatable within a constant expression, that function must be explicitly marked constexpr or, for an implicitly defined assignment operator, must satisfy certain requirements[11]. Building on this precedence of inferring an implicitly defined member function operator as constexpr when the enclosing class meets certain criteria, we propose that the function call operator (template for generic lambdas) be inferred as constexpr when it satisfies the requirements of a constexpr function. We also propose that for a generic lambda, the function call operator template be inferred as constexpr, unless no possible instantiation of that template would satisfy the requirements of a constexpr function.

Consider the following examples that illustrate when we would expect a lambda's call operator to be constexpr and when not:

```
struct NonLiteral { NonLiteral() { } };
strict Literal { };

auto L = [NL = NonLiteral{}] { return 0; };

constexpr int I = L();   // OK under this proposal.
                         // Closure type is non-literal, but function
                         // call operator satisfies the constraints of a
                         // constexpr function and so can be called in
                         // constexpr context.

auto L2 = [L = Literal{}] (int n) { return n + n; };

constexpr int J = L2(3); // OK under this proposal.

auto L3 = [](NonLiteral nl) { return 0; };
```

[11] N4431 12.8 [class.copy]/26 … The implicitly-defined copy/move assignment operator *[for a class X]* is constexpr if
— X is a literal type, and
— the assignment operator selected to copy/move each direct base class subobject is a constexpr function, and
— for each non-static data member of X that is of class type (or array thereof), the assignment operator selected to copy/move that member is a constexpr function.

```
constexpr int K = L3(NonLiteral{}); // NOT OK.

auto GL = [] { asm(""); return 0; };
constexpr int L = GL(); // NOT OK.
```

Additionally, if constexpr is explicitly specified, we propose that it be allowed either before or after the mutable specifier (taking into account the resolution to EWG 135[12]). Since we are proposing that lambda-expressions be allowed within *core-constant-expressions*, a closure object can be created during evaluation of a constant expression, and such closure objects should be allowed to have call operators that are both mutable and constexpr (hence one specifier should not exclude the other).

3.3.1 Why allow `constexpr` to be inferred for the function call operator?

Gabriel Dos Reis – one of the original designers of generalized constant expressions[13] – has voiced support for `constexpr` inference::

> *[14] [...] It is sad that we needed to introduce a keyword to get the notion accepted by the C++ community, and we couldn't just infer 'constexprness' from the (inline) definition and use context.*

> *[15] [...] As we have been warming up to more compile-time computations, I think it is becoming clearer and clearer that requiring constexpr is making not only the code more verbose, but also that failure to repeat syntactically the semantics information already in possession of the compiler is making it harder to smoothly develop new programming techniques and pattern around the language features and standard library facilities.*

And Richard Smith – also a constexpr pioneer[16] – has in turn cautioned us:

> *[17] [...] The constexpr keyword does have utility.*
>
> *It affects when a function template specialization is instantiated (constexpr function template specializations may need to be instantiated if they're called in unevaluated contexts; the same is not true for non-constexpr*

[12] http://www2.open-std.org/jtc1/sc22/wg21/docs/papers/2015/n4421.html#135
[13] http://www2.open-std.org/jtc1/sc22/wg21/docs/papers/2003/n1521.pdf
[14] https://groups.google.com/a/isocpp.org/d/msg/std-proposals/gEbulr5SPdc/2wi9Gy38Mr4J
[15] https://groups.google.com/a/isocpp.org/d/msg/std-proposals/gEbulr5SPdc/xcs8-hPXOZkJ
[16] http://www2.open-std.org/jtc1/sc22/wg21/docs/papers/2013/n3597.html
[17] https://groups.google.com/a/isocpp.org/d/msg/std-discussion/nrAu_YbCbYM/0eQsx6ip1DwJ

> *functions since a call to one can never be part of a constant expression). If we removed the meaning of the keyword, we'd have to instantiate a bunch more specializations early, just in case the call happens to be a constant expression.*
>
> *It reduces compilation time, by limiting the set of function calls that implementations are required to try evaluating during translation. (This matters for contexts where implementations are required to try constant expression evaluation, but it's not an error if such evaluation fails -- in particular, the initializers of objects of static storage duration.)*
>
> *It is also useful as a statement of intent: by marking a function as constexpr, you request that a compiler issues a diagnostic if it can easily see that a call to that function can never appear in a constant expression. There is a limited set of cases in which such a diagnostic is mandatory, and in the other cases it's a quality-of-implementation issue (but in practice compilers do a reasonable job of checking this). This statement of intent is also useful to human readers and maintainers of the code -- when modifying a function marked 'constexpr', you are aware that the function is intended to be used in constant expressions, so you know not to add (for instance) dynamic memory allocation to it, and if you do, the compiler stands a chance of telling you that you broke the users of your library. Obviously this checking is imperfect, because it doesn't provide a guarantee, but it still has some value.*
>
> *[18] [...] Today, that third-party code can move its constructor definition out of line, and the only impact that has on its consumers is a possible performance change. With your proposed change, the third-party library would *break source compatibility* every time it moved an inline function out of line, or otherwise changed the set of cases where that function could be part of a constant expression (for instance, if they added logging to the function).*
>
> *That's a horrible burden to impose on libraries. The status quo is that library authors have to opt into this compatibility burden by marking their functions as 'constexpr', and I think that's appropriate.*

Since Richard has informed at least one of the authors privately that he supports inference of `constexpr` for a lambda's function call operator – and since he was not pressed for his rationale (although we shall try and elicit one should he have the time, and should EWG require us to), we surmise this is probably because the issues having to do with inferring

[18] https://groups.google.com/a/isocpp.org/d/msg/std-proposals/nxP9zixxFmY/yXnwRgKM2g0J

`constexpr` for general functions are mostly mitigated by the requirement that the function call operator for *lambda-expressions* is inline.

Additionally, there is already precedent within C++14 for inference of `constexpr` when it comes to constructors and assignment operators (which have more involved rules for being `constexpr` than for regular functions).

3.4 The conversion function (to pointer-to-function) should, if it exists, be constexpr.

The conversion function of a lambda (with no *lambda-capture*) should always be constexpr, considering that it always returns a *constant expression* (the address of a function). If the function call operator of that lambda is constexpr, then its conversion function should return the address of a constexpr function. Consider:

```
// OK.  The conversion function is constexpr, and returns a constant
// expression. The function call operator is also constexpr.
constexpr int (*cfp)(int) = [](auto i) { return i; };

// The function call operator of this lambda can not be constexpr
// given a static local variable. But the conversion function still is.
constexpr int (*ncfp)(int) = [](auto i) { static int L; return i; };

static_assert(cfp(3) == 3, "");
static_assert(ncfp(3) == 3, ""); // Error. ncfp does NOT point to a
                                 // constexpr function.
// As another example ...
struct Literal { };
struct NonLiteral : Literal {
  NonLiteral() { };
};
auto GL = [](auto l) { return l; };

constexpr Literal (*cfp2)(Literal)        = GL;
constexpr NonLiteral (*ncfp2)(NonLiteral) = GL;

constexpr Literal Lit      = cfp2(Literal{});
constexpr NonLiteral NLit  = ncfp2(NonLiteral{});  // Error.
```

3.5 Lambda expressions in unevaluated operands are not being proposed

When the prototype implementation of `constexpr` lambdas[19] was reported, the question arose whether *lambda-expressions* should be allowed within unevaluated operands[20]. The

[19] https://github.com/faisalv/clang/tree/constexpr-lambdas

[20]

issues related to allowing *lambda-expressions* within unevaluated operands[21] (mangling within signatures etc.) are unrelated to this proposal and so will not be discussed further. Those interested are encouraged to explore the references in the afore-referenced footnotes.

4 Implementation

An implementation using clang has been prototyped on github[22] by one of the authors. Examples of code that the implementation is successfully able to compile is available for review[23] and a sample extended example is included in Appendix A.

As Richard Smith has outlined, there appear to be two approaches to implementing `constexpr`:

> [24]*[...] There are basically two different ways that people have historically implemented constant expression evaluation in C and C++ compilers:*
>
> *1) "fold": the AST is rewritten as constant expressions are simplified. In some implementations this happens as you parse, in others it happens as a separate step. So when you build a + operation whose operands are 1 and 1, you end up with an expression "2" and no evidence that you ever had a '+'. This also means you can use essentially the same code to perform various kinds of optimization.*
> *2) Real evaluation: the AST represents the code as written, and a separate process walks it and produces a symbolic value from it.*
>
> *Most implementations seem to do (1) in some way or another. [...]*
> *Clang has always done (2).*

Given the framework Richard Smith composed for `constexpr` and the foundation Doug Gregor orchestrated for lambda expressions, teaching clang how to evaluate lambda expressions during constant evaluation was fairly straightforward. The trickiest aspect was teaching the constant expression visitor for lambdas to deal with nested captures (especially the interleaving of reference captures with by-value captures) and array captures; while making sure each capture – during the compile-time execution of the function call operator – was substituted with the right value from the activations on the stack.

---

http://www.isocpp.org/forums/iso-c-standard-future-proposals?place=msg%2Fstd-proposals%2FWllUuPBDyxQ%2FVuZrxNxD8GgJ

[21] http://accu.org/cgi-bin/wg21/message?wg=core&msg=23350 / http://www2.open-std.org/jtc1/sc22/wg21/docs/cwg_defects.html#1607

[22] https://github.com/faisalv/clang/tree/constexpr-lambdas

[23] https://github.com/faisalv/clang/blob/constexpr-lambdas/test/CXX/clambdas/cxx1z-constexpr-lambdas.cpp

[24] https://groups.google.com/a/isocpp.org/d/msg/std-proposals/qcKUf-U7_YU/SRfxv76_ekkJ

5 Teachability

We expect that this feature would be easily teachable – imposing little, if any, additional burden – once students have been taught about `constexpr` and lambdas. There is no new syntax (unless you insist on annotating the function call operator as constexpr) so we expect that the use of constexpr lambdas by students should come naturally, should they find themselves having to perform compile time computations.

One might hypothesize that the introduction of constexpr lambdas to the language might facilitate beginner-to-intermediate students benefitting inadvertently from compile time computations in their code, without having to worry about the technicalities.

The current rules are easy to teach in the sense that it's easy to state that lambdas cannot be used in constant expressions, but it's rather harder to explain why. Advanced users tend to ask why such a prohibition is in place when their lambda bodies and captures would otherwise fulfill the requirements for a constexpr function and/or a literal type, and there's no good technical reason for it that can be easily explained.

With that in mind, if EWG still raises significant concerns about the teachability of this feature, we will attempt to address them in greater detail, in a future revision of this paper.

6 Core Wording

We shall await feedback from EWG before presenting core wording that would be appropriate for a CWG audience. However, we present a general strategy (based on the working paper N4296) hoping for CWG members who find themselves reading this section to provide us with early feedback should they identify fundamental flaws in our approach:

In [basic.types] 3.9/10.5.2:

> Allow closure types to be literal types if all of their non-static data members are of non-volatile literal types .

In [expr.prim.lambda] 5.1.2/1:

> Add the `constexpr`$_{opt}$ terminal to the *lambda-declarator* production (don't ignore EWG 135?)

In [expr.prim.lambda] 5.1.2/5:

> State that if the function call operator (member template) for lambdas satisfies the requirement of a constexpr function, it is constexpr.

In [expr.const] 5.20/2:

> Remove bullets 2.6 and 2.10 to allow *lambda-expressions* and evaluations within *lambda-expressions* within *core constant expressions*.

7 Acknowledgment

We would like to thank Richard Smith for all his contributions and comments (publicly and privately) in this space. We are also grateful to those users (such as Louis Dionne, Joel Falcou) who have struggled with the lack of this feature, and articulated support for it.

Appendix A

Sample Code Compiled by the Prototype Implementation

```
constexpr auto getFactorializer = [] {
  unsigned int optimization[6] = {};

  auto for_each = [](auto *b, auto *e, auto pred) {
    auto *it = b;
    while (it != e) pred(it++ - b);
  };

  for_each(optimization, optimization + 6, [&](int n) {
    if (!n) optimization[n] = 1;
    else
      optimization[n] = n * optimization[n-1];
  });

  auto FacImpl = [=](auto fac, unsigned n) {
    if (n <= 5) return optimization[n];
    return n * fac(fac, n - 1);
  };
  auto Fact = [=](int n) {
    return FacImpl(FacImpl, n);
  };
  return Fact;
};

constexpr auto Factorial = getFactorializer();

static_assert(Factorial(5) == 120, "");
static_assert(Factorial(10) == 3628800, "");
```