“Constexpr for the Elven-kings under the sky,
Templates for the Dwarf-lords in their halls of stone,
Runtime C++ for Mortal Men doomed to die,
Macros for the Dark Lord on his dark throne
In the Land of C++ where the Shadow worlds lie.
One Proposal to rule them all, One Proposal to find them,
One Proposal to bring them all, and in the standard bind them,
In the Land of C++ where the Shadow worlds lie.”
constexpr function parameters

2019-09-19

Uber
About me

- David Stone (david.stone@uber.com)
- Software engineer at Uber
- On C++ Standardization Committee
Agenda

01 How does constexpr work now?
02 What am I proposing?
03 Where is this in the C++ committee?
04 Related work
How constexpr works now
constexpr int x = 5;
int a[x] = {};
std::bitset<x> bits;
static_assert(x > 1);
constexpr double inverse(double x) {
    if (x == 0.0) {
        throw std::runtime_error("Ahh! Panic!!");
    }
    return 1.0 / x;
}
constexpr auto compile_error = inverse(0.0);
auto runtime_computation = inverse(value_from_file);
auto throws = inverse(0.0);
The difference

- constexpr variables require compile-time construction and destruction
- constexpr functions allow compile-time evaluation if everything they do is allowed at compile time
core constant expressions cannot...

- read values that are not constant expressions
- call functions that are not constexpr or not defined
- use volatile
- exceed implementation limits
- invoke undefined / indeterminate behavior
- throw
- inspect the common initial subsequence of a union
- reinterpret_cast or cast from void *
- do some things with addresses (more later)
Example

```cpp
constexpr int sometimes(bool b) {
    return b ? 6 : 1 / 0;
}
constexpr int a = sometimes(true); // 6
constexpr int b = sometimes(false); // ill-formed, diagnostic
int c = sometimes(true); // 6, may be evaluated at run time
int d = sometimes(false); // undefined behavior at run time
```
Proposal
Basic feature

```cpp
void f(constexpr int x) {
    static_assert(x == 5);
}
```
is_constant_expression

- Tells you whether an expression is a constant expression

static_assert(is_constant_expression("awesome"));
void f(constexpr int x) {
    static_assert(x == 5);
}

void f(int x) {
    assert(x == 5);
}
"Maybe constexpr"

void f(constexpr int x) {
  if constexpr (is_constant_expression(x)) {
    static_assert(x == 5);
  }
}

Design Goals

- Consistency
  - Eliminate metaprogramming
- Improve performance
- Improve diagnostics
Interlude: Discordia
No Shadow Worlds

- Segregated from the rest of the language
  - "Regular" run-time C++
  - constexpr functions
  - templates
  - macros
constexpr: the shadow of run-time C++

- **C++11**
  - No if
  - No for
  - Lots of recursive `condition ? expression : function(arg)`

- **C++17**
  - No memory allocation
Macros: the shadow of functions

- No loops
- No mutation
- (Almost) no recursion
- No scoping
- Weird if statements
Templates: the shadow of functions

- Compile-time parameters are lexically split off
  - f<0>(1)
- Seems fine
  - at first...
With a template, you cannot...

- pass template arguments to constructors
- pass template arguments to overloaded operators
- overload on whether a value is known at compile time
- put a compile-time argument after a run-time argument
- forward a pack of arguments, some of them compile time and some run time
There are a thousand hacking at the branches of evil to one who is striking at the root.

- Henry David Thoreau
When they first built the University of California at Irvine they just put the buildings in. They did not put any sidewalks, they just planted grass. The next year, they came back and put the sidewalks where the trails were in the grass.

- Larry Wall
Now (without this proposal)

auto a = std::array<int, 2>{};
a[0] = 1;
a[1] = 5;
a[2] = 3; // undefined behavior

auto t = std::tuple<int, std::string>{};
std::get<0>(t) = 1;
std::get<1>(t) = "asdf";
std::get<2>(t) = 3; // compile error
The future (with this proposal)

```cpp
auto a = std::array<int, 2>{};
a[0] = 1;
a[1] = 5;
a[2] = 3; // compile failure

auto t = std::tuple<int, std::string>{};
t[0] = 1;
t[1] = "asdf";
t[2] = 3; // compile failure
```
Now (without this proposal)

std::true_type{}
The future (with this proposal)

true
Now (without this proposal)

std::integral_constant<int, 24>{}
std::constant<int, 24>  // proposed
std::constant<24>  // also proposed
boost::hana::int_c<24>
boost::mpl::int_<24>
// A bunch more I don't know about
The future (with this proposal)
Now (without this proposal)

template<int n>
void f(boost::hana::int_<n>);
The future (with this proposal)

void f(constexpr int x);
Now (without this proposal)

do_math(boost::hana::int_c<8>);
The future (with this proposal)

do_math(8);
Now (without this proposal)

0 <=> 1 < 0;
// valid, as intended

0 <=> 1 < nullptr;
// valid, due to implementation detail
0 <=> 1 < 0;
// valid, as intended

0 <=> 1 < nullptr;
// error: no overloaded operator== comparing
// strong_equality with nullptr_t
Now (without this proposal)
c.f(n)(x)
Now (without this proposal)

c.template f<n>(x)
The future (with this proposal)

c.f(n, x)
Now (without this proposal)

auto const slow = std::regex("(A+|B+)*C");
The future (with this proposal)

```cpp
auto const fast = std::regex("(A+|B+)*C");
```
Now (without this proposal)

auto const glob = std::regex("*");  
// throws std::regex_error
The future (with this proposal)

auto const glob = std::regex("*");
// static_assert failed "Regular expression token
// '/*' must occur after a character to repeat."
bounded::integer

bounded::integer<0, 10> x(5);
bounded::integer<5, 6> y(6);
auto const z = x + y;
// decltype(z) == bounded::integer<5, 16>
// z == 11
Now (without this proposal)

using namespace bounded::literal;
auto a = bounded::integer<0, 10>(5_bi);
auto b = a + 5_bi;
auto c = bounded::integer<0, 10>(15); // Undefined behavior
The future (with this proposal)

auto a = bounded::integer<0, 10>(5);
auto b = a + 5;
auto c = bounded::integer<0, 10>(15); // Compile error
Now (without this proposal)

???
The future (with this proposal)

static_assert(pow(2_meters, 3) == 8_cubic_meters);
meters runtime_value;
std::cin >> runtime_value;
if (pow(runtime_value, 2) > 100_square_meters) {
    throw std::exception{};
}
The future (with this proposal)

```c
constexpr size_t strlen(constexpr char const * s) {
    for (char const * p = s; ; ++p) {
        if (*p == '\0') return static_cast<size_t>(p - s);
    }
}

size_t strlen(char const * s) {
    __asm__("SSE 4.2 insanity");
}
```
The future (with this proposal)

#include <emmintrin.h>
void f(int something, constexpr char y) {
    __m128i x;
    // ...
    // This intrinsic requires the second argument
    // to be a compile-time constant
    x = __builtin_ia32_aeskeygenassist128(x, y);
}
Why we need this proposal

- Uniform caller syntax
- Better run times
- Add compile-time diagnostics to libraries
- No need to change your callers
- Better compile times
- No need to reinvent the entire language in a metaprogramming library
Metaprogramming

- Powerful
- The wrong solution
How does it work?
Just like a template
What types can be used?

- **Literal types**
  - A constexpr constructor and destructor

- **Strong structural equality**
  - Defaulted operator `==`
  - All members have strong structural equality

- **Same as non-type template parameters**
Where can I use this?

```c++
void f(constexpr int x, array<int, x> const & a)
    noexcept(x > 5)
    requires x % 2 == 0
{
    static_assert(x == 6);
}
```
"Maybe constexpr"

- void f(constexpr int x)
- Use cases
  - Forward arguments
  - Two implementations that diverge in only one area
Controlled divergence

auto & operator[](constexpr size_t index) {
    if constexpr(is_constant_expression(index)) {
        static_assert(index < size());
    }
    return __data[index];
}
Overload resolution

- constexpr parameters are a tie-breaker
  - Matters only for otherwise ambiguous calls
  - Types must be the same
Standardization
Design by paper, design by committee

● Before each meeting, proposals are sent out
  ○ Nothing is discussed that does not have a paper

● Discuss papers between meetings via email list

● Discuss papers at meetings and vote
  ○ Everyone in the room gets one vote
  ○ SF | F | N | A | SA
  ○ Typically 2:1 in favor

● Sometimes, the design changes
Standardization Progress

● Discussed in EWG in 2018, June
  ○ Voted not to merge old version into C++20
  ○ Strong encouragement to pursue this direction
  ○ 13 | 14 | 15 | 3 | 1
Related work
consteval functions

● Added in C++20
  ○ Similar to constexpr
  ○ Requires the function to be evaluated at compile time
  ○ Similar meaning to constexpr on a variable

● Do not exist at run time
consteval functions

- Does not help if only some of the code is compile time
  - Cannot be used to implement tuple::operator[]
  - Cannot be used to add static_assert into array
  - Solves none of the motivating examples
- Cannot overload with run-time functions based on compile-timeness
- Cannot use their parameters as compile-time constants
is_constant_evaluated

- Many of the same problems as consteval
  - Requires the entire statement to be a compile-time constant
  - constexpr parameters gives fine-grained control
- Is determined by how the caller uses the result
constexpr double inverse(double x) {
    if (x == 0.0) {
        throw std::runtime_error("Ahh! Panic!!");
    }
    return 1.0 / x;
}

auto throws = inverse(0.0);
is_constant_evaluated

constexpr double inverse(double x) {
    if (x == 0.0) {
        if (is_constant_evaluated()) {
            ???
        }
        throw std::runtime_error("Ahh! Panic!!");
    }
    return 1.0 / x;
}

auto throws = inverse(0.0);
constexpr double inverse(constexpr double x) {
    if (x == 0.0) {
        if constexpr (is_constant_expression(x)) {
            static_assert(false);
        }
        throw std::runtime_error("Ahh! Panic!!");
    }
    return 1.0 / x;
}

auto compile_error = inverse(0.0);
Parametric expressions

- Proposal from Jason Rice: P1121
- Introduces the concept of "hygienic macros"
  - `using add(auto a, auto b) { return a + b; }`
  - Supports evaluating exactly once or 0-N times
- Proposes allowing constexpr parameters
  - Just to the parametric expressions
- Solves many problems
- Creates yet another shadow world
Parametric expressions

- Would not replace existing functions
  - Body is always inlined, like macros
  - Does not support overloading
  - Requires "deduction"
Syntax
"Maybe constexpr" syntax

- constexpr in this presentation
- Possibilities
  - constexpr?
"Maybe constexpr" syntax

- constexpr in this presentation
- Possibilities
  - constexpr?
"Maybe constexpr" syntax

- constexpr in this presentation
- Possibilities
  - constexpr?
  - maybe constexpr
Principles

- Consistency
## constexpr and consteval

<table>
<thead>
<tr>
<th></th>
<th>constexpr</th>
<th>consteval</th>
</tr>
</thead>
<tbody>
<tr>
<td>variables</td>
<td></td>
<td></td>
</tr>
<tr>
<td>functions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## constexpr and consteval

<table>
<thead>
<tr>
<th></th>
<th>constexpr</th>
<th>consteval</th>
</tr>
</thead>
<tbody>
<tr>
<td>variables</td>
<td>must be done at compile time</td>
<td></td>
</tr>
<tr>
<td>functions</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
# constexpr and consteval

<table>
<thead>
<tr>
<th></th>
<th>constexpr</th>
<th>consteval</th>
</tr>
</thead>
<tbody>
<tr>
<td>variables</td>
<td>must be done at compile time</td>
<td></td>
</tr>
<tr>
<td>functions</td>
<td>done at compile time if necessary, usable at compile time if possible</td>
<td></td>
</tr>
</tbody>
</table>
## constexpr and consteval

<table>
<thead>
<tr>
<th></th>
<th>constexpr</th>
<th>consteval</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>variables</strong></td>
<td>must be done at compile time</td>
<td>not supported</td>
</tr>
<tr>
<td><strong>functions</strong></td>
<td>done at compile time if necessary, usable at compile time if possible</td>
<td>must be done at compile time</td>
</tr>
</tbody>
</table>
# constexpr and consteval

<table>
<thead>
<tr>
<th></th>
<th>constexpr</th>
<th>consteval</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>variables</strong></td>
<td>done at compile time if necessary, usable at compile time if possible</td>
<td>must be done at compile time</td>
</tr>
<tr>
<td><strong>functions</strong></td>
<td>done at compile time if necessary, usable at compile time if possible</td>
<td>must be done at compile time</td>
</tr>
<tr>
<td>constexpr</td>
<td>consteval</td>
<td></td>
</tr>
<tr>
<td>-----------</td>
<td>----------</td>
<td></td>
</tr>
<tr>
<td>maybe compile time</td>
<td>definitely compile time</td>
<td></td>
</tr>
</tbody>
</table>
# constexpr and consteval

<table>
<thead>
<tr>
<th></th>
<th>constexpr</th>
<th>consteval</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>variables</strong></td>
<td>must be done at compile time</td>
<td>not supported</td>
</tr>
<tr>
<td><strong>functions</strong></td>
<td>done at compile time if necessary, usable at compile time if possible</td>
<td>must be done at compile time</td>
</tr>
</tbody>
</table>
## constexpr and consteval

<table>
<thead>
<tr>
<th>constexpr</th>
<th>consteval</th>
</tr>
</thead>
<tbody>
<tr>
<td>maybe compile time</td>
<td>definitely compile time</td>
</tr>
</tbody>
</table>
How do we want to program

- Are we Discordians
- Do we care about consistency?
Before I learned the art, a parameter was just a parameter, and an argument, just an argument. After I learned the art, a parameter was no longer a parameter, an argument, no longer an argument. Now that I understand the art, a parameter is just a parameter and an argument is just an argument.

- Bruce Lee (paraphrased)
Overload resolution

void f(int); // 1
void f(constexpr long); // 2

f(42); // calls 1
Overload resolution

void f(int); // 1
void f(constexpr int); // 2

f(42); // calls 2
Overload resolution

void f(unsigned); // 1
void f(constexpr long); // 2

f(42); // ambiguous