Document: P0596R1
Revises: P0596R0
Date: 10-05-2019
Audience: SG7
Authors: Daveed Vandevoorde (daveed@edg.com)

Side-effects in constant evaluation: Output and consteval variables

Revision history

R0	Initial revision proposing `std::constexpr_trace` and `std::constexpr_assert`.
R1	Rename `constexpr_trace` to `constexpr_report`. Add general treatment of side effects in constant evaluation. Introduce consteval variables.

Introduction

In early 2017, four papers were proposed with the goal of changing the course of "static reflection" for C++:

- P0598R0 ("Through Values Instead of Types"):
  This was the high-level proposal to change from representing reflections as types to representing them as values and using constexpr evaluation as the computational driver. The proposed direction was agreed upon and is now more concretely discussed in P1240R1 ("Scalable Reflection") and P1733R0 ("User-friendly and Evolution-friendly Reflection: A Compromise").
- P0597R0 ("`std::constexpr_vector<T>`"):
  This paper anticipated the need for dynamically-allocated structures at compile in order to support constexpr-based reflection. This paper morphed into something more general culminating in P0784R7 ("More constexpr containers") which is headed for C++20.
- P0596R0 ("`std::constexpr_trace` and `std::constexpr_assert`"):
  This paper anticipated that the increasing use of constexpr computation — particularly resulting from the direction suggested in P0598 — would lead to a need for compile-time output to (a) aid debugging and (b) allow reflection libraries to provide helpful diagnostics.
- P0595R0 ("The "`constexpr`" Operator"):
  This paper anticipated the need to allow the implementation of constexpr functions to adapt to whether they are invoked to produce a manifest constant expression or an ordinary expression that might be evaluated at compile time. Again, it was expected that this would be particularly

useful to some reflection application, and eventually the capability made it into the C++20 draft through P0595R2 ("`std::is_constant_evaluated`").

Three of the four papers have made significant progress either by becoming part of the proposed C++20 draft, or by becoming an evolving major proposal (P1240). The exception is P0596R0. At first, the feature seemed easy enough to specify and, indeed, it was the first proposal to be experimentally implemented. However, producing output during constant evaluation turns out to be tricky business because "output" is a *side effect* and constant evaluation in C++ never had to worry about that before. Furthermore, having made more progress with static reflection and reflective metaprogramming, we're finding that there are other kinds of side effects that we'd like to consider. Hence this paper which attempts to resolve the issues holistically.

In what follows we'll discuss:

- a revision of the feature proposed in the previous version of this paper (P596R0)
- a discussion of the side effects this introduces, their consequences, and proposed mitigation
- a new feature called "consteval variables" which runs into the same "side effect" issues and benefits from the same mitigation

Constexpr output

The original version of this paper proposed to special function templates:

```
namespace std {
  template<typename ... Ts> constexpr
    void constexpr_trace(Ts && ... values);
  template<typename ... Ts> constexpr
    void constexpr_assert(bool, Ts && ... values);
}
```

The first is meant to output the given values to "the console" if invoked "at compile time" or do nothing otherwise. The second is similar, except that the output of the values is only to be performed if the first argument is false, and the program is ill-formed in that case.

This paper keeps those proposed functions, except to rename the first one:

```
namespace std {
  template<typename ... Ts> constexpr
    void constexpr_report(Ts && ... values);
  template<typename ... Ts> constexpr
    void constexpr_assert(bool, Ts && ... values);
}
```

This capability was implemented in the EDG front end for *some* types (integers and character arrays) using the following intrinsics:

```
namespace std {
  constexpr void __report_constexpr_value(long long);
  constexpr void __report_constexpr_value(unsigned long long);
  constexpr void __report_constexpr_value(char const *ntbs);
  constexpr void __report_constexpr_value(char const *char_array,
                                          long long  n);
}
```

(The last variation outputs the first n characters pointed to by the first argument. This is useful, for example, to output `string_view`s.)

To illustrate the effect of these intrinsics, we can compile the following program:

```
#include <meta>  // To activate the intrinsics
constexpr int f1() { return 42; }
constexpr char const* f2() { return " is the answer"; }
consteval int g(int n) {
  std::__report_constexpr_value((long long)f1());
  std::__report_constexpr_value(f2());
  std::__report_constexpr_value("!!!!", n);
  return 42;
}
int main() {
  g(2);  // Two exclamation marks.
  g(0);  // No exclamation mark.
}
```

which produces (with the experimental implementation):

```
$ eccp --c++20 test.cpp
>> output from std::__report_constexpr_value
>> at line 11 of test.cpp
42 is the answer!!
>> end output from std::__report_constexpr_value

>> output from std::__report_constexpr_value
>> at line 12 of test.cpp
42 is the answer
>> end output from std::__report_constexpr_value
$
```

Note that because function `g` is a `consteval` function, the invocations of the intrinsics are always "manifestly constant-evaluated" and therefore the compile time output does happen[1]. If `consteval` is dropped this example produces no output. Even if `consteval` is replaced by `constexpr` this produces no output, because the invocations of `g` are not manifestly constant-evaluated.

Predictability

In the proposal above, conditioning the side effect of constexpr output on being "manifestly constant-evaluated" is meant to improve the predictability of that output. After all, for other expression evaluations we do not know whether the compiler will attempt constant evaluation.

Unfortunately, it turns out this is not sufficient in practice. In particular, we cannot always know whether an constant-expression is *actually* evaluated when dealing with template deduction. For example:

```
template<typename> constexpr int g() {
  std::__report_constexpr_value("in g()\n");
  return 42;
}
template<typename T> int f(T [g<T>()]);  // (1)
template<typename T> int f(T*);          // (2)
int r = f<void>(nullptr);
```

When the compiler resolves the call to `f` in this example, it substitutes `void` for `T` in both declarations (1) and (2). However, for declaration (1), it is unspecified whether `g<void>()` will be invoked: The compiler may decide to abandon the substitution as soon as it sees an attempt to create "an array of `void`" (in which case the call to `g<void>` is *not* evaluated), or it may decide to finish parsing the array declarator and evaluate the call to `g<void>` as part of that.

We can think of a few realistic ways to address/mitigate this issue:

1. Make attempts to trigger side-effects in expressions that are "tentatively evaluated" (such as the ones happening during deduction) ill-formed with no diagnostic required (because we cannot really require compilers to re-architect their deduction system to ensure that the side-effect trigger is reached).
2. Make attempts to trigger side-effects in expressions that are "tentatively evaluated" cause the expression to be non-constant. With our example that would mean that even a call `f<int>(nullptr)` would find (1) to be nonviable because `g<int>()` doesn't produce a constant in that context.
3. Introduce a new special function to let the programmer control whether the side effect takes place anyway. E.g., `std::is_tentatively_constant_evaluated()`. The specification work for this is probably nontrivial and it would leave it unspecified whether the call to `g<void>` is evaluated in our example.

[1] We're proposing non-normative encouragement to produce a "diagnostic message" (see [defns.diagnostic]).

We propose to follow option 2. Option 3 remains a possible evolution path in that case, but we prefer to avoid the resulting subtleties if we can get away with it.

There is another form of "tentative evaluation" that is worth noting. Consider:

```
constexpr int g() {
  std::__report_constexpr_value("in g()\n");
  return 41;
}
int i = 1;
constexpr int h(int p) {
  return p == 0 ? i : 1;
}
int r = g()+h(0);  // Not manifestly constant-evaluated but
                   // g() is typically tentatively evaluated.
int s = g()+1;     // To be discussed.
```

Here `g()+h(0)` is *not* a constant expression because `i` cannot be evaluated at compile time. However, the compiler performs a "trial evaluation" of that expression to discover that. In order to comply with the specification that `__report_constexpr_value` only produce the side effect if invoked as part of a "manifestly constant-evaluated expression", two implementation strategies are natural:

1. "Buffer" the side effects during the trial evaluation and "commit" them only if that evaluation succeeds.
2. Disable the side effects during the trial evaluation and repeat the evaluation with side effects enabled if the trial evaluation succeeds.

The second option is only viable because "output" as a side effect cannot be *observed* by the trial evaluation. However, further on we will consider another class of side effects that *can* be observed within the same evaluation that triggers them, and thus we do not consider option 2 a viable general implementation strategy.

The first option is more generally applicable, but it may impose a significant toll on performance if the amount of side effects that have to be "buffered" for a later "commit" is significant.

An alternative, therefore, might be to also consider the context of a non-constexpr variable initialization to be "tentatively evaluated" and deem side-effects to be non-constant in that case (i.e., the same as proposed for evaluations during deduction). In the example above, that means that `g()+1` would *not* be a constant expression either (due to the potential side effect by `__report_constexpr_value` in an initializer that is allowed to be non-constant) and thus `s` would not be statically initialized.

Consteval variables

Consteval functions were added to C++20 as a kind of function that *must* produce constant results. However, we can also look at them from a different perspective:

Ordinary functions	Constexpr functions	Consteval functions
Only “exist” at run time (target machine)	“Exist” at compile time (host machine) *and* run time (target machine)	Only “exist” at compile time (host machine)

With that in mind, we can look at the situation of variables in the same way:

Ordinary variables	Constexpr variables	Consteval variables?
Only “exist” at run time (target machine)	“Exist” at compile time (host machine) *and* run time (target machine)	Only “exist” at compile time (host machine)

Note that the fact that *constexpr* variables are immutable (i.e., implicitly `const`) falls out naturally from this view: Since such a variable lives in both domains we cannot reasonably order mutations of those variables. If constexpr variables were mutable, consider what would happen to the following:

```
constexpr int v = 42;
extern void f(int *p);
int main() {
  f(&v);                // Modifies p?
  constexpr int r = v;  // ???
  return r;
}
```

The call to `f` could modify `v` but the compiler would have no way to reflect that mutation in `r`.

In this point of view, consteval variables are a natural closure of our entity system: They are variables that *only* exist at compile time (i.e., in the constant-evaluation machine). Unlike constexpr variables, however, consteval variables need *not* be immutable because they need not be viewed consistently from the two domains (run time and compile time). Instead, they have constraints that parallel somewhat those of consteval functions:

- they can only be mutated during manifest constant evaluation, and
- their address cannot “leak” into the run time.

Furthermore:

- consteval variables have no linkage (because the ODR semantics would be a little odd otherwise)
- consteval variables can only be declared in namespace scope (local consteval variables seem confusing; consteval static data members seem reasonable, except for the ODR semantics)
- their destructors are evaluated at the end of unit translation (in reverse order of construction)
- consteval variables cannot be references (to avoid difficult questions wrt. object lifetimes)

An example hopefully illustrates the potential power of this feature:

```
consteval int counter = 0;
consteval int counter_value() { return counter++; }
int r1 = counter_value();  // Statically initializes r1 to 0.
void f() {
  if (counter_value() == 2) {
    ... dead code because counter_value() produced 1, not 2.
  }
}
static_assert(counter_value() == 2);  // Okay.
int main() {
  return counter_value();  // Returns the constant value 3.
}
```

This example involves a simple compile-time integer state that persists and mutates during compilation. However, considerably more complex variables can be envisioned. For example, in reflective metaprogramming, we expect it to be useful to create registries that can be updated incrementally throughout a translation unit, and, for example, have the destructor act on the aggregated registered information.

Of particular relevance to this paper, here, is that *mutation of consteval variables is a side-effect*!

This leads to the "predictability" considerations that we discussed for constexpr output:

- what happens if consteval variable mutation occurs during deduction?
- what happens if consteval variable mutation occurs during the trial evaluation of a potentially constant variable initializer?

We propose that the answers be the same as for the constexpr output case: Attempting consteval variable mutation in these "tentative" contexts simply fails to constant-evaluate.

For example:

```
consteval int v = 0;
```

```
constexpr int r = v++;  // Okay.  No “trial” since the variable
                        // is constexpr.  r =0 0 and v == 1.
int e = v++;  // Error.  The evaluation of v++ fails during trial
              // evaluation.  So this is not manifestly constant-
              // evaluated, and consteval variable mutation is not
              // permitted in outside such contexts.
```

Another example, this time illustrating the effect on deduction (i.e., SFINAE):

```
template<typename> consteval int cnt = 0;
template<int> struct X {};
template<typename T> int f(X<++cnt<T>>*) { return 1; }
template<typename T> int f(T*) { return 2; }
int r = f<int>(nullptr);  // Initializes r to 2.
```

When the compiler considers the first candidate `f`, it attempts to evaluate `++cnt<int>`, but since that occurs as part of deduction, the side effect does not take place and constant evaluation fails. So that candidate is dropped, and the second candidate remains (with no problems substituting int for `T` in`T*`).

Suggested questions/polls

1. Do we agree with a general policy of limiting constant-evaluation side effects to manifestly constant-evaluated expressions that are in no way tentative (not part of SFINAE, not trial evaluations for initializers, and may other potential tentative evaluation contexts)?

2. Do we want something like `std::constexpr_report` and `std::constexpr_assert`?
   2.1. If so, do we want the suggested parameterization?
   2.2. If so, do we want the suggested names?

3. Do we want something like consteval variables?
   3.1. If so, do we agree with the suggested outline for their constraints and capabilities?