Document	P1331R0
Date	2019-01-04
Author	CJ Johnson <johnsoncj@google.com>
Audience	Evolution Working Group (EWG)

Permitting trivial default initialization in constexpr contexts

Preface

This proposal is intended to augment and work alongside P0784 [1]. It should be thought of as a form of bug fix as it makes the features in P0784 more complete for C++2a.

Introduction

This paper proposes permitting default initialization of trivially default constructible types in constexpr contexts while continuing to disallow the invocation of undefined behavior. In short, so long as uninitialized values are not read from, such states should be permitted in constexpr in both heap and stack allocated scenarios.

Prerequisite

Let the below `NontrivialType` and `TrivialType` type definitions be imported and visible for all following example code snippets.

```
struct NontrivialType {
  bool val = false;
};
```

```
struct TrivialType {
  bool val;
};
```

Justification as of C++17

For types with non-trivial default constructors, default initialization is defined as calling the default constructor. This behavior is consistent across runtime and constexpr contexts (assuming the default constructor is marked as or defaulted as constexpr). Unfortunately, this guarantee does not generically apply across the language. For types with a trivial default constructor, default initialization does not compile in constexpr contexts. This becomes a pain point when writing generic code. Template instantiations that are well defined in constexpr for types with user-defined constexpr default constructors will fail to compile when instantiated with, for example, any of the fundamental integer types provided by the language.

The below code example, `Example1`, shows how this inconsistency can negatively affect code that one would expect to be consistent across template instantiations. If this proposal is adopted, all four below callsites, instead of just the first three, would be valid and compile covering the full matrix of runtime/constexpr by trivially/non-trivially default constructible.

```
namespace Example1 {

template <typename T>
constexpr T f(const T& other) {
  T t;
  t = other;
  return t;
}
```

```
// These three successfully compile
auto nontrivial_rt = f(NontrivialType{});
auto trivial_rt = f(TrivialType{});
constexpr auto nontrivial_ct = f(NontrivialType{});

// As of C++17, this fails to compile meaning `Example1::f(const T&)` is not
// generic
constexpr auto trivial_ct = f(TrivialType{});

}  // `namespace Example1`
```

Note: `nontrivial_rt` is constant initialized since `Example1::f<NontrivialType>(...)` is valid in constexpr. As a bonus, by making `Example1::f<TrivialType>(...)` valid in constexpr, `trivial_rt` will also be given constant initialization.

Justification for C++2a

In addition to the unexpectedly inconsistent behavior demonstrated above, the presence of P0784 [1] in C++2a strengthens the need for this proposed change. Uninitialized memory in the form of default initialized trivially default constructible values is available, but only when heap allocated. This proposal would simply fix the "bug" disallowing that same behavior for stack allocated variables.

The below examples, `Example2` and `Example3`, show side-by-side what should be semantically equivalent functions. The current C++2a draft standard, with the adoption of P0784, would permit, in constexpr contexts, the function definitions that use heap allocation but not the function definitions that use stack allocation. This proposal, if adopted with P0784, would result in all four below function callsites being valid.

```
namespace Example2 {

template <typename T>
constexpr T f1(const T& other) {
  auto* t = new T;
  *t = other;
  T out = *t;
  delete t;
  return out;
}

template <typename T>
constexpr T f2(const T& other) {
  T t;
  t = other;
  T out = t;
  return out;
}

// Successfully compiles
constexpr auto trivial_ct_h = f1(TrivialType{});

// Fails to compile meaning `Example2::f1(const T&)` and
// `Example2::f2(const T&)` are observably different
constexpr auto trivial_ct_s = f2(TrivialType{});

}  // `namespace Example2`
```

```
namespace Example3 {

template <typename T>
constexpr T f1(const T& other) {
  auto* t = new T[1];
  t[0] = other;
  T out = t[0];
  delete[] t;
```

```
  return out;
}

template <typename T>
constexpr T f2(const T& other) {
  T t[1];
  t[0] = other;
  T out = t[0];
  return out;
}

// Successfully compiles
constexpr auto trivial_ct_h = f1(TrivialType{});

// Fails to compile meaning `Example3::f1(const T&)` and
// `Example3::f2(const T&)` are observably different
constexpr auto trivial_ct_s = f2(TrivialType{});

}  // `namespace Example3`
```

On "Constexpr all the things!"

`absl::InlinedVector<T, N, A>` is a `std::vector<T, A>`-like type that takes advantage of the Small Buffer Optimization (SBO). SBO is performed by including an inlined buffer inside the type. For sufficiently small instances of InlinedVector, no allocation is performed. In such cases, SBO is a performance win when reading the elements (due to cache locality) and when initializing instances (due to a lack of a need to allocate a dynamically sized buffer). At runtime, as an optimization, unused bytes in the inlined space are left uninitialized. "Don't pay for what you don't use," as they say.

In the spirit of P0784 [1], if InlinedVector were to be made available in constexpr, the behavior should be consistent with the behavior of runtime instances. The unused inlined bytes should be able to be left uninitialized both so that implementers may be able to expose accidental uninitialized reads in their implementations and so that there is the *potential* to incur less cost at compile time as a result of creating such a buffer inside the type.

Avoiding undefined behavior

As always, undefined behavior cannot be invoked in constexpr. In order to allow trivial default initialization, some mechanism must be in place that prevents reading uninitialized values. The mechanism should continue to be compilation failure; however, instead of the overly restrictive current behavior (failing to compile at the point of initialization), the compiler should only fail at the point in code where the uninitialized read takes place.

This should not be difficult for compiler developers to implement nor should it cause much if any compile time overhead. The steps required are similar to detecting reads of out-of-lifetime variables, something that current compilers already implement for constant evaluation. [2]

Open questions

- Reading uninitialized instances of `unsigned char` at runtime is not undefined. Does that imply it should be a special case in constexpr as well? In other words, should the standard allow `unsigned char a; auto b = a;` in constexpr?

  > The most reasonable thing would be to say that (in the short term at least), reading the value of an uninitialized object is non-constant regardless of its type. We can revisit that decision if we find a use case, but going in the other direction would be more problematic due to being a potentially breaking change. [3]

References

[1] P0784: More constexpr containers

[2] Personal correspondence with Richard Smith about this proposal

[3] Comment by Richard Smith in [RFC] P1331: Permitting trivial default initialization in constexpr contexts