`constexpr` cast from `void*`: towards `constexpr` type-erasure

Document #:	P2738R0
Date:	2022-12-15
Programming Language C++	
Audience:	EWG
Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>
	David Ledger <davidledger@live.com.au>

Abstract

We propose to allow a limited form of casting from `void*` to support type erasure in `constexpr`.

Revisions

R0

Initial revision

Motivation

Using the proposed feature, we were able to get `std::format` work at compile time. Other standard facilities could be made `constexpr` such as `std::function_ref`, `std::function`, and `std::any`.

Storing `void*` instead of a concrete type is a commonly used compilation firewall technique to reduce template instantiations and the number of symbols in compiled binaries. For example, with a template class that stores a `T*` pointer to some data, member functions are instantiated for each template combination; if instead some member functions along with a `void*` were part of a non-template base class, those member functions are instantiated only once.

On memory constrained embedded platforms, a common approach to achieve run-time memory savings is to ensure common code paths. Type erasure is helpful to achieve common code paths. To save memory, it is desirable to evaluate code at compile time to the maximal extent. To keep code common between compile time and run-time, limited type erasure is required at compile time.

Vocabulary types like the proposed `std::function_ref` communicate constraints to the caller much better than an unconstrained callable. Communication of constraints on arguments is not exclusively beneficial to the run-time domain.

This type erasure allows more of the standard library (such as `std::format`) to be made `constexpr`, and by lifting restrictions it promotes unification of `constexpr` and non-`constexpr` implementations.

The following example, courtesy of Jason Turner, illustrates the use of this feature

```
#include <string_view>

struct Sheep {
    constexpr std::string_view speak() const noexcept { return "Baaaaaa"; }
};

struct Cow {
    constexpr std::string_view speak() const noexcept { return "Mooo"; }
};

class Animal_View {
    private:
    const void *animal;
    std::string_view (*speak_function)(const void *);

    public:
    template <typename Animal>
    constexpr Animal_View(const Animal &a)
    : animal{&a}, speak_function{[](const void *object) {
            return static_cast<const Animal *>(object)->speak();
    }} {}

    constexpr std::string_view speak() const noexcept {
        return speak_function(animal);
    }
};

// This is the key bit here. This is a single concrete function
// that can take anything that happens to have the "Animal_View"
// interface
std::string_view do_speak(Animal_View av) { return av.speak(); }

int main() {
    // A Cow is a cow. The only think that makes it special
    // is that it has a "std::string_view speak() const" member
    constexpr Cow cow;
    // cannot be constexpr because of static_cast
    [[maybe_unused]] auto result = do_speak(cow);
    return static_cast<int>(result.size());
}
```

Note: The above example while functionally similar to `virtual` based polymorphism has a key difference, it does not require a vtable pointer to be carried around with all the objects that conform to the interface.

Facilities like `format` and `function_ref` use a similar pattern.

Design

We propose to allow casting from a pointer of `void*` to a pointer of type `T` in `constexpr` if the type of the object at that address is exactly of type `T`. In particular, we are not proposing allowing conversion to a pointer that would be interconvertible, let alone unrelated. Indeed, most `constexpr` evaluator implementations are based on value, rather than memory, and anything that would require reinterpreting the object as another type is generally not possible.

However, most implementations have a way to know the type of an object pointed to by a given pointer (for diagnostics, `constexpr` allocation, virtual dispatch, or other reasons), and so a cast from `void*` to a pointer of the type of the pointed to object is implementable.

Do we want to support conversion to pointer to base?

At first approach, it would make sense to support casts to base classes. After all, casting to a base class is possible in `constexpr` contexts. However, in a non-`constexpr` context, consider

```
struct A {
    virtual void f() {};
    int a;
};
struct B {
    int b;
};
struct C: B, A {};

int main() {
    C c;
    void* v = &c;
    assert(static_cast<B*>(v) == static_cast<B*>(static_cast<C*>(v))); // #1
}
```

`#1` does not hold. Both expressions return different addresses. So casting to a derived class then its base is not isomorphic to casting to the base directly so, for consistency and simplicity we are not proposing to cast to allow cast from `void*` to base classes either.

Implementation Experience

Clang

Implementing this in Clang was trivial. Indeed, clang does support `constexpr` conversions from `void*` but limits their use to inside of `std::allocator::allocate`, as part of the `constexpr` allocation machinery. Lifting that restriction doesn't present any particular challenge.

The other clang `constexpr` interpreter (based on bytecode) also tracks the origin of pointers and can implement this proposal.

MSVC & GCC

Front-end engineers from both GCC and MSVC indicated this proposal offered no particular implementation challenge as their respective implementations already track the full type information of all pointers.

EDG

Someone from EDG indicated this proposal should be implementable, albeit with performance inefficiencies. Their implementation does not track enough information for the type to be immediately available and they would have to reconstruct it by walking the AST of the enclosing whole object.

Impact on future implementations

This constrains an evaluator to know about the type of an object a pointer points to. Adding this information to an implementation that does not have it could be challenging in the future, and, as more implementation start looking at interpreting `constexpr` code, knowing they must preserve that information will inform their design. We should therefore do that change sooner than later to guarantee it remains implementable.

Wording

� Constant expressions [expr.const]

An expression E is a *core constant expression* unless the evaluation of E, following the rules of the abstract machine, would evaluate one of the following:

[...]

- a conversion from type *cv1* `void*` to a pointer-to-object type `T` unless E evaluates to the address of an object of type *cv2* `T` where *cv2* is the same cv-qualification as, or greater cv-qualification than, *cv1*.
- a `reinterpret_cast`;
- a modification of an object unless it is applied to a non-volatile lvalue of literal type that refers to a non-volatile object whose lifetime began within the evaluation of E;

Feature test macro

[Editor's note: In `[tab:cpp.predefined.ft]`, bump the value of`__cpp_constexpr` to the date of adoption] .

Acknowledgments

Thanks to Jason Merril, Daveed Vandevoorde and Cameron DaCamara for their input in the implementability of this feature. Thanks to Jason Turner, Michael Caisse and Brian Bi for providing feedbacks and encouraging this work.

References

[N4892] Thomas Köppe *Working Draft, Standard for Programming Language C++*
https://wg21.link/N4892