Changing the active member of a union inside constexpr

Document #:	P1330R0
Date:	2018-11-10
Project:	Programming Language C++
Audience:	EWG, CWG

Reply-to:	Louis Dionne <ldionne@apple.com>
	David Vandevoorde <daveed@edg.com>

1 Abstract

We propose allowing the active member of a union to be changed inside a constant expression, such that the following code is valid:

```
union Foo {
  int i;
  float f;
};

constexpr int use() {
  Foo foo{};
  foo.i = 3;
  foo.f = 1.2f; // not valid today, valid with this paper
  return 1;
}

static_assert(use());
```

2 Motivation

[P0784] greatly expands what can be done inside a constant expression to allow memory allocations, amongst other things. The explicit goal of [P0784] is to allow standard containers to work inside constant expressions. However, some containers like `std::string` and `std::optional` use unions inside their implementations. Since changing the active element of a union is not currently allowed inside a constant expression, those containers can't be implemented properly with [P0784] alone. The goal of this paper is to serve as (hopefully) the last stepping stone for `std::string` being constexpr-friendly.

3 Implementability concerns

The implementation difficulty related to this proposal is that any member of a union inside the constexpr evaluator needs a field saying whether it is the active member of the union, and that field needs to be updated whenever the active member of the union changes. However, any references to that member of the union also have to be updated so that the constexpr evaluator can catch undefined behavior when accessing the union through references to a now inactive member. Depending on the implementation strategy used by the compiler, this could mean varous levels of implementation difficulty. Here's an extract of an email exchange between Louis Dionne and David Vandevoorde discussing the issue:

Louis Dionne writes:

> David Vandevoorde writes:
>
> > You have to think in terms of catching undefined behavior (since constexpr evaluation has to do that). A constexpr evaluator has to maintain a hidden "field" that describes the current active member. That "field" is part of the union object representation of the evaluator. So, presumably assigning a new value to the union object can update that part of the union object representation itself. However, if you have an lvalue for a member of the union (an int, say), it's not 100% clear that you know that that int is a union subobject; if you don't have a reliable way to reach that union representation, you cannot update its active field.
> >
> > Our implementation _can_ do it and so can Clang, I believe (because all their addresses — pointers and glvalues — are represented through symbolic paths), but I'm not 100% sure that there aren't interesting (e.g., more efficient) implementation strategies that wouldn't be able to do this.
>
> I think I understand the problem. Basically, you need to update all "references" to that union subobject to say "this isn't the active member of the union anymore" so you can detect the UB in case the program tries to access that member of the union. It's easy when the implementation uses symbolic paths, because then you just update the original member to say "this isn't the active member of the union anymore", and all references to it will automatically see the change. However, if an implementation used a different strategy, the union may need something like a list of all the references to the active member so that you can go and update them if the active member of the union changes.

We think that current implementations can handle that without too many problems, but this needs to be discussed with more implementers.

4 Another (not entirely related) problem

While surveying the implementation of `std::string` in libc++, I found the following piece of code that is used to check whether the string being stored is short or long (simplified example):

```
template <typename CharT>
struct basic_string {
  struct LongRepresentation {
    std::size_t cap_;
    std::size_t size_;
    char* data_;
  };

  struct ShortRepresentation {
    unsigned char size_;
    char data_[SSO_SIZE];
  };

  union Representation {
    LongRepresentation long_;
    ShortRepresentation short_;
  };

  Representation repr_;

  bool is_long() const {
    // access possibly inactive member of the union
    return bool(repr_.short_.size_ & SHORT_MASK);
  }
};
```

Put simply, we always reserve the very first bit of the string (whether short or long) to store whether the string is short or long. To check whether the string is long or short, we always access that first bit through the short string representation, which may be inactive if we're actually holding a long string. My understanding is that this is already undefined behavior today: this is not covered by [**class.mem**] **10.3/25** because `LongRepresentation` and `ShortRepresentation` do not share a common initial subsequence.

Unless I'm mistaken, libc++ needs to fix this problem regardless of whether this paper is adopted, and in fact adopting this paper is orthogonal to that. It may just meant that making `std::string` constexpr will be a challenge for libc++ due to unfortunate implementation choices. Note that libstdc++'s implementation of `std::string` does not suffer from the same problem.

However, if I am wrong and libc++ does not have UB today, then [**expr.const**] **7.7/2.8** would break `std::string` inside constexpr:

> An expression `e` is a *core constant expression* unless the evaluation of `e`, following the rules of the abstract machine (6.8.1), would evaluate one of the following expressions:

  - an lvalue-to-rvalue conversion (7.3.1) that is applied to a glvalue that refers to a non-active member of a union or a subobject thereof;

5 Proposed wording

This wording is based on the working draft [N4762].

Strike **[expr.const] 7.7/2.10**:

> An expression `e` is a *core constant expression* unless the evaluation of `e`, following the rules of the abstract machine (6.8.1), would evaluate one of the following expressions:
>
> - `this` (7.5.2), except in a constexpr function or a constexpr constructor that is being evaluated as part of `e`;
>   [...]
> - ~~an assignment expression (7.6.18) or invocation of an assignment operator (10.3.6) that would change the active member of a union;~~

6 References

[N4762] Richard Smith, *Working Draft, Standard for Programming Language C++*
`http://www.open-std.org/jtc1/sc22/wg21/docs/papers/2018/n4762.pdf`

[P0784] David Vandevoorde & al, *More constexpr containers*
`http://wg21.link/P0784`