Updated wording and implementation experience for P1481 (constexpr structured bindings)

Document #:	P2686R0
Date:	2022-10-15
Programming Language C++	
Audience:	EWG
Reply-to:	Corentin Jabot <corentin.jabot@gmail.com>

Abstract

P1481R0 [1] proposed to allow reference to constant expressions to be themselves constant expressions, as a means to support `constexpr` structured bindings. This paper reports implementation experience on this proposal and provides updated wording.

Context

The context for this paper can be found in P1481R0 [1]. I was not aware to reach the original author, nor do I have the possibility to reproduce the original paper.

The gist of it is that the original author proposed to support `constexpr` structured binding by making

```
constexpr auto[a] = std::tuple(1);
```

Equivalent to

```
constexpr auto __sb = std::tuple(1);
const int& __a = std::get<0>(__sb);
```

Additional motivation

In addition to the original motivation, if we believe structured bindings are useful (they are, great feature!) and we also believe in constexpr (as a means to increase type safety, improve runtime performance, etc), then both features ought to work together.

In addition to that, Expansion Statements (P1306R1 [2]) aim to add a new kind of for loop with the express purpose to loop over tuples at compile time.

```
auto tup = std::make_tuple(0, ‘’a, 3.14);
// ill-formed without this paper
template for (constexpr auto [idx, member] : std::views::enumerate(meta::data_members_of(^T)) )
    fmt::print("{} {}", idx, foo.[:member:]);
```

History

Interestingly, this paper was last seen in Kona in 2020. The concerns were

- Lack of implementation
- It was presented late in the C++20 cycle

> Encourage further work on this proposal
>
> | SF | F | N | A | SA |
> |---|---|---|---|---|
> | 9 | 16 | 4 | 0 | 0 |

This paper thereby provides an implementation. I've also update the wording as CWG rewrote the impacted section, and added the wording to support the constexpr keyword on structured bindings declarations.

Implementation

Circle

Circle implements constexpr structured bindings - and generally supports initializing references with constant expressions, and Sean Baxter was not aware that the standard didn't support it. Sean further observed that this is a core language syntactic sugar and as such, users could expect it to work everywhere.

Clang

I implemented a prototype implementation in the hope to weed out issues. It is available on Compiler Explorer.

Please note that by lack of time, I have not yet published the last version of the implementation, but that should hopefully be done before Kona.

I don't think the implementation revealed particular issues (my own inaptitudes non-withstanding), I, however, believe `[basic.odr]` might need to be tweaked.

> A variable `x` that is named by a potentially-evaluated expression `E` is odr-used by `E` unless x is a reference that is usable in constant expressions ([expr.const]).

I don't think this is sufficient. Consider for example,

```
void foo() {
    const int a = 1;
    const int& b = a;
    auto l = [] { return b; }; // we should not capture b implicitly here,
                               // even if b is usable in constant expressions
}
```

In my prototype, I check that the initializer of the reference is itself a constant expression, and that seems to work.

Wording

� Constant expressions [expr.const]

A variable is *potentially-constant* if it is constexpr or it has reference or const-qualified integral or enumeration type.

A constant-initialized potentially-constant variable V is *usable in constant expressions* at a point P if V's initializing declaration D is reachable from P and

- V is constexpr or it is of reference type initialized with a core constant expression,
- V is not initialized to a TU-local value, or
- P is in the same translation unit as D.

An object or reference is *usable in constant expressions* if it is

- a variable that is usable in constant expressions, or
- a template parameter object, or
- a string literal object, or
- a temporary object of non-volatile const-qualified literal type whose lifetime is extended to that of a variable that is usable in constant expressions, or
- a non-mutable subobject or reference member of any of the above.

� Structured binding declarations [dcl.struct.bind]

A structured binding declaration introduces the *identifier*s v_0, v_1, $\text{v}_2, \ldots$ of the *identifier-list* as names of *structured binding*s. Let *cv* denote the *cv-qualifier*s in the *decl-specifier-seq* and *S* consist of the `constexpr` and *storage-class-specifier*s of the *decl-specifier-seq* (if any). A *cv* that includes `volatile` is deprecated; see **??**. First, a variable with a unique name *e* is introduced. If the *assignment-expression* in the *initializer* has array type *cv1* `A` and no *ref-qualifier* is present, *e* is defined by *attribute-specifier-seq*$_{opt}$ *S cv* `A` *e* ;
and each element is copy-initialized or direct-initialized from the corresponding element of the *assignment-expression* as specified by the form of the *initializer*. Otherwise, *e* is defined as-if by *attribute-specifier-seq*$_{opt}$ *decl-specifier-seq ref-qualifier*$_{opt}$ *e initializer* ;
where the declaration is never interpreted as a function declaration and the parts of the declaration other than the *declarator-id* are taken from the corresponding structured binding declaration. The type of the *id-expression* *e* is called `E`. [*Note:* `E` is never a reference type. — *end note*]

If the *initializer* refers to one of the names introduced by the structured binding declaration, the program is ill-formed.

If `E` is an array type with element type `T`, the number of elements in the *identifier-list* shall be equal to the number of elements of `E`. Each v_i is the name of an lvalue that refers to the element i of the array and whose type is `T`; the referenced type is `T`. [*Note:* The top-level cv-qualifiers of `T` are *cv*. — *end note*] [*Example:*

```
        auto f() -> int(&)[2];
        auto [ x, y ] = f();            // x and y refer to elements in a copy of the array return
value
        auto& [ xr, yr ] = f();         // xr and yr refer to elements in the array referred to
by f's return value
```

— *end example*]

� The `constexpr` and `consteval` specifiers [dcl.constexpr]

The `constexpr` specifier shall be applied only to the definition of a variable or variable template, a structured binding, or the declaration of a function or function template. The `consteval` specifier shall be applied only to the declaration of a function or function template. A function or static data member declared with the `constexpr` or `consteval` specifier is implicitly an inline function or variable. If any declaration of a function or function template has a `constexpr` or `consteval` specifier, then all its declarations shall contain the same specifier.

Feature test macros

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

Acknowledgments

We would like to thank Bloomberg for sponsoring this work. Thanks to Nicolas Lesser for the original work on P1481R0 [1].

References

[1] Nicolas Lesser. P1481R0: constexpr structured bindings. https://wg21.link/p1481r0, 1 2019.

[2] Andrew Sutton, Sam Goodrick, and Daveed Vandevoorde. P1306R1: Expansion statements. https://wg21.link/p1306r1, 1 2019.

[N4885] Thomas Köppe *Working Draft, Standard for Programming Language C++* https://wg21.link/N4885