`constexpr` structured bindings

and

references to `constexpr` variables

Document #:	P2686R2
Date:	2023-09-14
Programming Language C++	
Audience:	EWG, CWG

Reply-to:	Corentin Jabot <corentin.jabot@gmail.com> Brian Bi <bbi10@bloomberg.net>

Abstract

P1481R0 [2] proposed allowing references 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.

Revisions

Revision 2

We provide wording for option 3 (symbolic addressing), which is the direction chosen by EWG in Varna. We also allow `constinit` structured bindings, mostly because we could not find a good reason not to, and we think it's best to avoid too many exceptions and inconsistencies.

Revision 1

After core expressed implementability concerns of the original design as it pertains to onstexpr references to automatic storage duration variables, we provide different options.

Revision 0

Design and wording soimilar to that of P1481R0 [2].

Issues with R0 and possible solutions

The previous revision of this paper, (P2686R0 [1]), was approved by the EWG in Issaquah and was subsequently reviewed by CWG, which found the proposed wording to be quite insufficient.

No issue arises with allowing `constexpr` structured binding in general, except for the case of an automatic storage duration structured binding initialized by a tuple, i.e.,

```
void f() {
    constexpr auto [a] = std::tuple(1);
    static_assert(a == 1);
}
```

which translates to

```
void f() {
    constexpr auto __sb = std::tuple(1);  // __sb has automatic storage scenario.
    constexpr const int& a = get<0>(__sb);
}
```

When the structured binding is over an array or a class type, it doesn't create actual references, so we have no issue. When the structured binding is not at function scope, the underlying tuple object has static storage duration, and its address is a permitted result of a constant expression.

So the problematic case occurs when we are creating an automatic storage duration (i.e., at block scope) structured binding of a tuple (or *tuple-like*) object. This specific situation, though, is not uncommon.

The initial wording simply allowed references initialized by a constant expression to be usable in constant expressions. This phrasing failed to observe that the address of a `constexpr` variable with automatic storage duration may be different for each evaluation of a function and, therefore, cannot be a *permitted result of a constant expression*.

The CWG asks that the EWG consider and pick one direction to resolve these concerns. Some options are explored below.

Possible solutions

0. Allowing static and non-tuple `constexpr` structured binding

We should be clear that nothing prevents `constexpr` structured bindings from just working when binding an aggregate or an array since those are modeled by special magic aliases that are not quite references (which allows them to work with bitfields).

A `constexpr` structured binding of a tuple *with static storage duration*, i.e.,

```
static constexpr auto [a, b] = std::tuple{1, 2};
```

would also simply work as it would be equivalent to

```
static constexpr auto __t = std::tuple{1, 2};
static constexpr auto & a = std::get<0>(__t);
static constexpr auto & b = std::get<1>(__t);
```

Supporting this solution requires no further changes to the language than basically allowing the compiler to parse and apply the `constexpr` specifier. Independently of the other solutions presented here, this option would be useful and should be done.

The problematic scenario is an automatic storage duration binding to a `tuple`.

We could stop there, not try to solve this problem, and force users to use `static`. We would, however, have to ensure that expansion statements work with static variables since that was one of the motivations for this paper.

1. Making `constexpr` implicitly static

We could make `constexpr` variables implicitly static, but doing so would most certainly break existing code, in addition to being inconsistent with the meaning of `constexpr`:

```
int f() {
    constexpr struct S {
        mutable int m ;
    } s{0};
    return ++s.m;
}

int main() {
    assert(f() + f() == 2); // currently 2. Becomes 3 if 's' is made implicitly static
}
```

So this solution is impractical. We could make `constexpr` static only in some cases to alleviate some of the breakages or even make only `constexpr` bindings static, not other variables, but this option feels like a hack rather than an actual solution.

2. Always re-evaluate a call to `get`?

We could conceive that during constant evaluation, tuple structured bindings are replaced by a call to `get` every time they are constant-evaluated. This would help with `constexpr` structured binding but would still disallow generic cases:

```
constexpr in not_a_sb =1;
constexpr const int&  a = sb;
```

Additionally, this would be observable in scenarios in which `get` would perform some kind of compile-time i/o such as proposed by P2758R0 [3].

3. Symbolic addressing

The most promising option — the one we think should be pursued — is for `constexpr` references to designate a specific object, rather than an address, and to retain that information across constant evaluation contexts. This is how constant evaluation of references works, but this information is not currently persisted across constant evaluation, which is why we do not permit `constexpr` references to refer to objects with automatic storage duration (or subobjects thereof).

To quote a discussion on the reflector:

> This would also resolve a longstanding complaint that the following is invalid:
>
> ```
> void f() {
>     constexpr int a = 1;
>     constexpr auto *p = &a;
> }
> ```
>
> It seems like a lot of C++ developers expect the declaration of p to be valid, even though it's potentially initialized to a different address each time f is invoked.

This solution has the benefit of not being structured-binding specific and would arguably meet user expectations better than the current rule. Interestingly and maybe counter-intuitively, the constexprness of pointers and references is completely orthogonal to that of their underlying object:

```
int main() {
    static int i = 0;
    static constexpr int & r = i; // currently valid

    int j = 0;
    constexpr int & s = j; // could be valid under the "symbolic addressing" model
}
```

References can be constant expressions because we can track during constant evaluation which objects they refer to, independently of whether the value of that object is or isn't a constant expression.

We would have to be careful about several things. Pointers and references to variables with automatic storage duration cannot be used outside of the lifetime of their underlying objects, so they could not appear

- in template arguments
- as the initializer of a variable with static storage duration

Similarly, we can construct an automatic storage duration `constexpr` reference to a static variable but not a static `constexpr` reference bound to an automatic storage duration object.

Additional considerations

Thread-local variables

Taking the address of a thread-local variable may initialize the variable, and that initialization may not be a constant expression. Supporting references/pointers to thread-local variables would therefore require additional consideration, and we would probably want to allow it only if it were already initialized on declaration.

We could exclude thread locals from the design entirely as we're not sure a compelling use case exists for constexpr references to thread-local objects.

Lambda capture of `constexpr` references bound to automatic storage duration objects

`constexpr` references are not ODR-used. Therefore, a constexpr reference used in a lambda does not trigger a capture. This would be problematic for references bound to automatic storage duration objects:

```
auto f() {
    int i = 0;
    constexpr const int & ref = i;
    return [] {
        return ref;
    });
}
f(); //# ! try to access i outside of its lifetime
```

We will have to modify [basic.def.odr]/p5.1 so that `constexpr` references to automatic storage duration variables (or subobjects thereof) are ODR-used.

Wording for Option 3 (symbolic addressing)

� One-definition rule [basic.def.odr]

[*Editor's note:* Modify p5 as follows:]

A variable is named by an expression if the expression is an *id-expression* that denotes it. A variable x that is named by a potentially-evaluated expression ~~E~~ N that appears at a point P is *odr-used* by ~~E~~ N unless N is an element of the set of potential results of an expression E, where

- x is a reference that is usable in constant expressions [expr.const], or
- x is a variable of non-reference type that is usable in constant expressions and has no mutable subobjects, and E is an element of the set of potential results of an expression of non-volatile-qualified non-class type to which the lvalue-to-rvalue conversion [conv.lval] is applied, or
- x is a variable of non-reference type, and E is an element of the set of potential results of a discarded-value expression [expr.context] to which the lvalue-to-rvalue conversion is not applied.
- x is usable in constant expressions at P ([expr.const]) and E designates a reference that is constexpr-representable at P, or

- x is usable in constant expressions at P, E designates an object that is usable in constant expressions at P, and the lvalue-to-rvalue conversion ([conv.lval]) is applied to E, or
- E is a discarded-value expression ([expr.context]) to which the lvalue-to-rvalue conversion is not applied.

� Static initialization [basic.start.static]

Drafting note: The constant initialization of a variable implicitly includes the constant initialization of any temporary objects whose lifetimes are extended to that of the variable. All references to constant initialization from elsewhere in the standard currently refer only to variables with constant initialization. Removing the words "or temporary object" from this paragraph simplifies the wording elsewhere by avoiding the need to define when an object (as opposed to variable) is constant-initialized.

[*Editor's note:* Modify p2 as follows:]

Constant initialization is performed if a variable ~~or temporary object~~ with static or thread storage duration is constant-initialized[expr.const]. If constant initialization is not performed, a variable with static storage duration[basic.stc.static] or thread storage duration[basic.stc.thread] is zero-initialized[dcl.init]. Together, zero-initialization and constant initialization are called *static initialization*; all other initialization is *dynamic initialization*. All static initialization strongly happens before[intro.races] any dynamic initialization. [*Note:* The dynamic initialization of non-block variables is described in [basic.start.dynamic]; that of static block variables is described in [stmt.dcl]. — *end note*]

� Constant expressions [expr.const]

Drafting note: P0784R7 [5] abolished the previous restriction that constexpr constructors of non-literal class types may not be invoked during constant evaluation. The current wording of [expr.const]/2 still contains a special exception that allows a variable to be considered constant-initialized even though the initialization would invoke such a constructor; that wording is unnecessary since P0784R7 [5] was accepted.

Drafting note: A structured binding is a named lvalue, but is not a reference in the non-tuple-like cases; therefore, the current rules regarding references that are not usable in constant expressions ([expr.const]/8) do not always apply to structured bindings. The intent of the below wording is that structured bindings should be subject to the same restrictions during constant evaluation that would apply if they were references.

Drafting note: The definition of "constexpr-referenceable" below is written under the assumption that temporary objects are considered to have the storage duration described in CWG1634 [4], namely, that a temporary object whose lifetime is extended inherits the storage duration of the reference that is bound to it, and any other temproary object has a distinct storage duration.

Certain contexts require expressions that satisfy additional requirements as detailed in this subclause; other contexts have different semantics depending on whether or not an expression satisfies these requirements. Expressions that satisfy these requirements, assuming that copy elision[class.copy.elision] is not performed, are called *constant expressions*. [*Note:* Constant expressions can be evaluated during translation. — *end note*]

> *constant-expression:*
> > *conditional-expression*

[*Editor's note:* Insert a paragraph after p1:]

The *constituent values* of an object o are the value of o if it has scalar type and the values of any of o's subobjects of scalar type, other than inactive union members and subobjects thereof. The *constituent references* of an object o are the non-static data members of reference type of o and of any of o's subobjects that are neither inactive union members nor subobjects thereof.

[*Editor's note:* Insert a paragraph after p1:]

The constituent values and constituent references of a variable `x` are defined as follows:

- If `x` declares an object, the constituent values and references of that object are constituent values and references of `x`.
- If `x` declares a reference, that reference is a constituent reference of `x`.

For any constituent reference `r` of a variable `x`, if `r` is bound to a temporary object or subobject thereof whose lifetime is extended to that of `r`, the constituent values and references of that temporary object are also constituent values and references of `x`. This rule applies recursively.

[*Editor's note:* Insert a paragraph after p1:]

An object o is *constexpr-referenceable* from a point P if

- o has static storage duration, or
- o has automatic storage duration, and, letting `v` denote the variable corresponding to o's complete object, or the variable to whose lifetime that of o is extended, `v`'s smallest enclosing non-block scope and P's smallest enclosing non-block scope are the same function parameter scope.

[*Editor's note:* Insert a paragraph after p1:]

An object or reference `x` is *constexpr-representable* at a point P if, for each constituent value of `x` that points to or past an object o, and for each constituent reference of `x` that refers to an object o, o is constexpr-referenceable from P.

[*Editor's note:* Modify p2 as follows:]

A variable ~~or temporary object o~~ v is *constant-initialized* if

- either it has an initializer or its default-initialization results in some initialization being performed, ~~and~~

- the full-expression of its initialization is a constant expression when interpreted as a *constant-expression* ~~, except that if o is an object, that full-expression may also invoke constexpr constructors for o and its subobjects even if those objects are of non-literal class types.~~ [*Note:* ~~Such a class can have a non-trivial destructor.~~ Within this evaluation `std::is_constant_evaluated()` [meta.const.eval] returns `true`. — *end note*] , and
- immediately after the initializing declaration of `v`, the object or reference declared by `v` is constexpr-representable.

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

[*Editor's note:* Modify p4 as follows:]

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,
- V is not initialized to a TU-local value, or
- P is in the same translation unit as D.

An object or reference x is *usable in constant expressions* at point P if it is

- a variable that is usable in constant expressions, or
- a template parameter object [temp.param] or subobject thereof, or
- a string literal object [lex.string] or subobject thereof, or
- a temporary object of non-volatile const-qualified literal type whose lifetime is extended [class.temporary] to that of a variable that is usable in constant expressions, or
- a non-mutable subobject or reference member of any of the above.
- one of the following:
  - a non-volatile variable that is usable in constant expressions at P,
  - a temporary object of non-volatile const-qualified literal type whose lifetime is extended ([class.temporary]) to that of a variable that is usable in constant expressions at P,
  - a non-mutable subobject of any of the above, or
  - a reference member of any of the above

  that is constexpr-representable at P, subject to the restriction that if x is an object, it has no mutable subobjects.

[*Note:* An object can be usable in constant expressions even if its complete object o is disqualified from being usable in constant expressions because o has a mutable subobject. — *end note*]

[*Editor's note:* Add after p8:]

For the purposes of determining whether an expression is a core constant expression, the evaluation of an *id-expression* that names a structured binding `v` ([dcl.struct.bind]) has the following semantics:

- If `v` is an lvalue referring to the object bound to an invented reference `r`, the behavior is as if `r` were nominated.
- Otherwise, if `v` names an array member or class member, the behavior is that of evaluating e`[`i`]` or e`.`m, respectively, where e is the name of the variable initialized from the initializer of the structured binding declaration, and i is the index of the element referred to or m is the name of the member referred to by `v`, respectively.

[*Editor's note:* Modify p13 as follows:]

A *constant expression* is either a glvalue core constant expression that refers to ~~an entity that is a permitted result of a constant expression (as defined below)~~ an object or a non-immediate function, or a prvalue core constant expression whose value satisfies the following constraints:

- if the value is an object of class type, each non-static data member of reference type refers to an entity that is a permitted result of a constant expression,
- if the value is an object of scalar type, it does not have an indeterminate value[basic.indet],
- if the value is of pointer type, it contains the address of an object with static storage duration, the address past the end of such an object[expr.add], the address of a non-immediate function, or a null pointer value,
- if the value is of pointer-to-member-function type, it does not designate an immediate function, and
- if the value is an object of class or array type, each subobject satisfies these constraints for the value.
- each constituent reference refers to an object or a non-immediate function,
- no constituent value of scalar type is an indeterminate value ([basic.indet]),
- no constituent value of pointer type is a pointer to an immediate function or an invalid pointer value ([basic.compound]), and
- no constituent value of pointer-to-member type designates an immediate function.

An entity is a *permitted result of a constant expression* if it is an object with static storage duration that either is not a temporary object or is a temporary object whose value satisfies the above constraints, or if it is a non-immediate function. [*Note:* A glvalue core constant expression that either refers to or points to an unspecified object is not a constant expression. — *end note*]

� Declarations [dcl.dcl]

� Preamble [dcl.pre]

[*Editor's note:* Change p6 as follows:]

A *simple-declaration* with an *identifier-list* is called a *structured binding declaration* [dcl.struct.bind]. Each *decl-specifier* in the *decl-specifier-seq* shall be `constexpr`, `constinit`, `static`, `thread_local`, `auto` [dcl.spec.auto], or a *cv-qualifier*. [*Example:*

```
template<class T> concept C = true;
C auto [x, y] = std::pair{1, 2};    // error: constrained placeholder-type-specifier
// not permitted for structured bindings
```

— *end example*]

� Structured binding declarations [dcl.struct.bind]

[*Editor's note:* Change p1 as follows:]

A structured binding declaration introduces the *identifiers* $\mathtt{v}_0$, $\mathtt{v}_1$, $\mathtt{v}_2, \ldots$ of the *identifier-list* as names of *structured bindings*. Let *cv* denote the *cv-qualifiers* in the *decl-specifier-seq* and *S* consist of ~~the *storage-class-specifiers* of the *decl-specifier-seq* (if any)~~ each *decl-specifier* of the *decl-specifier-seq* that is `constexpr`, `constinit`, or a *storage-class-specifier* . A *cv* that includes `volatile` is deprecated; see [depr.volatile.type]. 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 [expr.prop]. — *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 $\mathtt{v}_i$ 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]

[*Editor's note:* Change p1 as follows:]

The `constexpr` specifier shall be applied only to the definition of a variable or variable template, a structured binding declaration, 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 [dcl.inline]. If any declaration of a function or function template has a `constexpr` or `consteval` specifier, then all its declarations shall contain the same specifier.

[...]

[*Editor's note:* Modify p6 as follows:]

A `constexpr` specifier used in an object declaration declares the object as const. Such an object shall have literal type and shall be initialized. In any `constexpr` variable declaration, the full-expression of the initialization shall be a constant expression [expr.const]. A `constexpr` variable that is an object, as well as any temporary to which a `constexpr` reference is bound, shall have constant destruction. Immediately after the initializing declaration of a `constexpr` variable `v`, the object or reference declared by `v` shall be constexpr-representable.

[*Example:*

```
struct pixel {
    int x, y;
};
constexpr pixel ur = { 1294, 1024 };     // OK
constexpr pixel origin;                  // error: initializer missing
```

— *end example*]

� The `constinit` specifier [dcl.constexpr]

Drafting note: Unlike in [dcl.constexpr], we don't need an explicit rule about the object or reference being constexpr-representable in this section, because the restriction added to [expr.const]/2 will cause the variable to have dynamic initialization if the object or reference is not constexpr-representable.

[*Editor's note:* Modify p1 as follows:]

The `constinit` specifier shall be applied only to a declaration of a variable with static or thread storage duration or to a structured binding declaration ([dcl.struct.bind]). If the specifier is applied to any declaration of a variable, it shall be applied to the initializing declaration. No diagnostic is required if no `constinit` declaration is reachable at the point of the initializing declaration.

� Template non-type arguments [temp.arg.nontype]

[*Editor's note:* Modify [temp.arg.nontype]/p1 as follows:]

A *template-argument* for a non-type *template-parameter* with declared type `T` shall be such that the invented declaration

```
T x = template-argument ;
```

satisfies the semantic constraints for the definition of a `constexpr` variable with static storage duration [dcl.constexpr]. If ~~the type~~ `T` ~~of a *template-parameter* [temp.param]~~ contains a placeholder type ([dcl.spec.auto]) or a placeholder for a deduced class type ([dcl.type.class.deduct]), the type of the parameter is ~~the type deduced for the variable `x` in the invented declaration~~ deduced from the above declaration.

```
T x = template-argument ;
```

~~If a deduced parameter type~~ If the parameter type thus deduced is not permitted for a *template-parameter* declaration [temp.param], the program is ill-formed.

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 Nina Dinka Ranns, Pablo Halpern, and Joshua Berne for their feedback.

Thanks to Richard Smith for the original discussion of possible solutions on the Core reflector and for feedback on the wording.

Thanks to Nicolas Lesser for the original work on P1481R0 [2].

References

[1] Corentin Jabot. P2686R0: Updated wording and implementation experience for p1481 (constexpr structured bindings). https://wg21.link/p2686r0, 10 2022.

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

[3] Barry Revzin. P2758R0: Emitting messages at compile time. https://wg21.link/p2758r0, 1 2023.

[4] Richard Smith. CWG1634: Temporary storage duration. https://wg21.link/cwg1634, 3 2013.

[5] Daveed Vandevoorde, Peter Dimov, Louis Dionne, Nina Ranns, Richard Smith, and Daveed Vandevoorde. P0784R7: More constexpr containers. https://wg21.link/p0784r7, 7 2019.

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