`constexpr` structured bindings

and

references to `constexpr` variables

Document #:	P2686R5
Date:	2024-11-12
Programming Language C++	
Audience:	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 5

- Fix examples in the core wording.
- Add notes on implementation

Revision 4

The definitions of ”constituent values” and ”constituent references” have been tweaked slightly with no change in meaning. The imprecise ”at a point [...] namespace scope” wording has been changed to refer to the nearest following point at namespace scope. The example in the wording that states which variables are constexpr-referenceable has been updated to clarify which variables are not constexpr-referenceable. A new term has been introduced in order to clarify the presentation of the definition of ”usable in constant expressions”. A new term has been introduced in order to reduce duplicative specification between ”constant-initialized” and the requirements for `constexpr` variable declarations. Function parameter scopes introduced by `requires` expressions are now ignored. Additional examples have been added to [expr.const] demonstrating the semantics of structured bindings during constant evaluation.

Remove the feature test macro, as core did not find a use case for you and recent papers bump both `__cpp_constexpr` and `__cpp_structured_bindings`.

This revision was approved by CWG in St Louis and is waiting an implementation to be forwarded to plenary.

Revision 3

CWG pointed out that the restrictions on volatile variables and mutable subobjects in the definition of "usable in constant expressions" have undesirable consequences. Those restrictions have therefore been moved to [basic.def.odr]. An unintentional omission in R2 has also been fixed: a local static constexpr reference shall not refer to a variable with automatic storage duration. Examples have also been added to the wording section.

Following discussion in EWG and on the reflector, we have also refined our explanation for why the proposed changes do not allow constexpr references declared inside lambdas to bind to variables with automatic storage duration declared in an enclosing function.

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 constexpr references to automatic storage duration variables, we provide different options.

Revision 0

Design and wording similar 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 [4].

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.

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.

Lambdas that reference automatic storage duration objects from the enclosing function

`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();
```

The current rules never require odr-use of `constexpr` references because any use of a `constexpr` reference can always be evaluated as if it were replaced by its initializer. However, in the above example, such a transformation is not desirable since it would require `i` to be captured. Instead, we need to modify [basic.def.odr]/p5.1 so that `constexpr` references to automatic storage duration variables (or subobjects thereof) are ODR-used. In the above example, `ref` therefore needs to be captured.

Because lambdas can name variables declared in an enclosing function, they also present other questions about the scope of this proposal. During EWG review in Kona (November 2023), examples similar to the following were discussed:

```
auto f1() {
    int i = 0;
    constexpr const int & ref = i;
    return [&] {
        constexpr const int & ref2 = i;
        return ref2;
    }();
}
auto f2() {
    int i = 0;
    constexpr const int & ref = i;
    return [&] {
        constexpr const int & ref2 = i;
        return ref2;
    };
}
```

In such examples, the relationship between the lifetime of the reference and that of the variable it refers to is not obvious, and we initially viewed this as a rationale to forbid such examples. However, further discussion on the reflector revealed that there are currently two different rules in the language that make the above examples ill-formed:

1. A `constexpr` reference may not refer to `i` because `i` has automatic storage duration. This restriction is found in [expr.const]/p14 and it is the one that we propose to relax in order to support constexpr structured bindings at block scope.
2. In a *lambda-expression*, any expression that names a variable with automatic storage duration declared in an enclosing function and that odr-uses that variable is not a constant expression. This restriction is found in [expr.const]/p5.13.

The second restriction follows from the fact that the use of an automatic variable from an

enclosing function has the same semantics as if it referred to a member of the closure type, e.g.,

```
auto f1() {
    int i = 0;
    constexpr const int & ref = i;
    class __lambda {
        int& __i;
        __lambda(int& i) : __i(i) {}
        auto operator()() const {
            constexpr const int & ref2 = *this.__i;
            return ref2;
        }
    };
    return __lambda(i)();
}
auto f2() {
    int i = 0;
    constexpr const int & ref = i;
    class __lambda {
        int& __i;
        __lambda(int& i) : __i(i) {}
        auto operator()() const {
            constexpr const int & ref2 = *this.__i;
            return ref2;
        }
    };
    return __lambda(i);
}
```

In the above ""desugared lambdas"", the `*this` always refers to an object of type `__lambda` whose lifetime begins before that of `ref2`. Consequently, [expr.const]/p9 applies: during the checking of whether `ref2`'s initializer is a constant expression, `*this` is treated as referring to an unspecified object, and so is `*this.__i`. Since P2280R4 [3], a reference to an unspecified object can participate in constant evaluation in limited ways, but cannot, of course, be the final result of a constant expression.

In the desugared forms, it is clear why the `ref2` cannot be constexpr in `f1` and `f2`. In current C++, the validity of a `constexpr` variable declaration (or any expression that is manifestly constant-evaluated) cannot depend on a control flow analysis that determines all paths by which that construct is reached; therefore, `*this.__i` must be rejected as the initializer of a `constexpr` variable as it cannot be *locally* verified that it refers to an entity known at compile time. We do not propose to introduce such control flow analysis to C++ as would be necessary to lift the restrictions in [expr.const]/p9 and [expr.const]/p5.13.

Other cases that are not allowed

We also do not propose to make the following code well-formed:

```
int& foo(int x) { return x; }
```

```
int bar() {
    constexpr int& r = foo(1);
}
```

On a callee-destroy implementation, the lvalue result of `foo(1)` is dangling because the parameter object `x` is destroyed immediately after `foo` returns to its caller. On a caller-destroy implementation, the parameter object `x` is not destroyed until after `r` is initialized. Therefore, on a callee-destroy implementation, the initialization of `r` has undefined behavior and is, for that reason, not a constant expression, while on a caller-destroy implementation, the only reason why this code is currently ill-formed is that `foo(1)` does not have static storage duration. Although we propose to allow `constexpr` references to refer to objects with automatic storage duration, which includes parameter objects, we do not think it is desirable for the above code to be well-formed only on caller-destroy implementations, as it would create a new portability issue with no known upside. Therefore we propose the restriction that when the referent of a `constexpr` reference has automatic storage duration, the referent must have the same innermost enclosing function parameter scope as the reference.

Teachability of the relaxed restrictions

Concerns were raised on the reflector about how to explain the changes proposed by this paper to the broader C++ community. In a few sentences, the changes could be explained as follows:

> You can now declare structured bindings `constexpr`. Because structured bindings behave like references, `constexpr` structured bindings are subject to similar restrictions as `constexpr` references, and supporting this feature required relaxing the previous rule that a `constexpr` reference must bind to a variable with static storage duration. Now, `constexpr` references and structured bindings may also bind to a variable with automatic storage duration, but only when that variable has an address that is constant relative to the stack frame in which the reference or structured binding lives.

For a C++ programmer who is curious about the rationale for the “constant relative to the stack frame” restriction as it disallows a `constexpr` reference in a lambda from binding to a local variable in an enclosing function, we suggest the following brief explanation:

> In a lambda, when you refer to a local variable `x` from an enclosing function, the compiler transforms that access into something like `(*this).__x`, where `__x` represents the captured address of `x`. The expression `(*this).__x` is not a constant expression because it isn’t known at compile time what object `this` points to.

Implenentation

This paper was implemented in EDG (Thanks Daveed). The implementation is available on Compiler explorer.

Note that EDG (and MSVC) is not capable of handling some valid C++23 code, where a temporary is persisted via lifetime extension in aggregate initialization.

```
struct A {
    int* const& r;
};
static int x;
static constexpr A a = {&x};
static_assert(a.r == &x);
```

[Compiler Explorer].

And this proposal make this problem somewhat more visible.

After discussion with EDG, we agree that this is an orthogonal issue that might require a core issue in the future if the implementation challenges prove too great.

Wording for Option 3a (symbolic addressing with the same-function restriction)

[basic.def.odr] One-definition rule

[*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

- x is a reference that is usable in constant expressions at P [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.

  Drafting note: Ideally, x should be allowed to have mutable subobjects as long as we don't touch the mutable parts of x. This would probably only require slightly more complex wording, but isn't in scope for this paper because the status quo also doesn't allow it.

- N is an element of the set of potential results of an expression E, where
  - E is a discarded-value expression ([expr.context]) to which the lvalue-to-rvalue conversion is not applied or
  - x is a non-volatile object that is usable in constant expressions at P and has no mutable subobjects and

- E is a class member access expression ([expr.ref]) naming a non-static data member of reference type and whose object expression has non-volatile-qualified type or
- the lvalue-to-rvalue conversion ([conv.lval]) is applied to E and E has non-volatile-qualified non-class type.

[*Example:*

```
int f(int);
int g(int&);
struct A {
    int x;
};
struct B {
    int& r;
};
int h(bool cond) {
    constexpr A a = {1};
    constexpr const volatile A& r = a;  // odr-uses a
    int _ = f(cond ? a.x : r.x);  // does not odr-use a or r
    int x, y;
    constexpr B b1 = {x}, b2 = {y};  // odr-uses x and y
    int _ = g(cond ? b1.r : b2.r);  // does not odr-use b1 or b2
    int _ = ((cond ? x : y), 0);  // does not odr-use x or y
    return [] {
        return b1.r;  // error: b1 is odr-used here because the object
                      // referred to by b1.r is not constexpr-referenceable
                      // here
    }();
}
```

— end example]

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 [6] 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 [6] 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 [5], 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

- if o has scalar type, the value of o;
- otherwise, the constituent values of any direct subobjects of o other than inactive union members.

The *constituent references* of an object o are

- any direct members of o that have reference type, and
- the constituent references of any direct subobjects of o other than inactive union members.

[*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, 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,

  the smallest scope enclosing v and the smallest scope enclosing P that are neither
  - block scopes nor
  - function parameter scopes associated with a *requirement-parameter-list*

  are the same function parameter scope.

[*Example:*

```
struct A {
    int m;
    const int& r;
};
void f() {
    static int sx;
    thread_local int tx;  // tx is never constexpr-referenceable
    int ax;
    A aa = {1, 2};
    static A sa = {3, 4};
    // The objects sx, ax, and aa.m, sa.m, and
    // the temporaries to which aa.r and sa.r are bound, are
    // constexpr-referenceable.
    auto lambda = [] {
        int ay;
        // The objects sx, sa.m, and ay (but not ax or aa), and the
        // temporary to which sa.r is bound, are constexpr-referenceable.
    };
}
```

— end example]

[*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*~~ *constant-initializable* 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 x declared by v is constexpr-representable, and
- if x has static or thread storage duration, x is constexpr-representable at the nearest point whose immediate scope is a namespace scope that follows the initializing declaration of v.

[*Editor's note:* Insert a paragraph after p2:]

A constant-initializable variable is *constant-initialized* if either it has an initializer or its default-initialization results in some initialization being performed.

[*Example:*

```
void f() {
    int ax = 0;                 // ax is constant-initialized
    thread_local int tx = 0;    // tx is constant-initialized
    static int sx;              // sx is not constant-initialized
    static int& rss = sx;       // rss is constant-initialized
    static int& rst = tx;       // rst is not constant-initialized
    static int& rsa = ax;       // rsa is not constant-initialized
    thread_local int& rts = sx; // rts is constant-initialized
    thread_local int& rtt = tx; // rtt is not constant-initialized
    thread_local int& rta = ax; // rta is not constant-initialized
    int& ras = sx;              // ras is constant-initialized
    int& rat = tx;              // rat is not constant-initialized
    int& raa = ax;              // raa is constant-initialized
}
```

— *end example*]

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 *potentially usable in constant expressions* at point P if it is

- the object or reference declared by a 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 template parameter object [temp.param],
- a string literal object [lex.string],
- a non-mutable subobject of any of the above, or
- a reference member of any of the above.

An object or reference x is *usable in constant expressions* at point P if it is an object or reference that is potentially usable in constant expressions at P and is constexpr-representable at P.

- a variable that is usable in constant expressions, or
- a template parameter object [temp.param], or
- a string literal object [lex.string], 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 at P, or
- a non-mutable subobject or reference member of any of the above.

[*Example:*

```
struct A {
    int* const & r;
};
void f(int x) {
    constexpr A a = {&x};
    static_assert(a.r == &x);  // OK
    [&] {
        static_assert(a.r != nullptr);  // error: a.r is not usable in
                                        // constant expressions at this point
    }();
}
```

— end example]

[*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 element 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 by v or m is the name of the member referred to by v, respectively.

[*Example:*

```
#include <tuple>
struct S {
    mutable int m;
    constexpr S(int m): m(m) {}
    virtual int g() const;
};
void f(std::tuple<S&> t) {
    auto [r] = t;
    static_assert(r.g() >= 0);  // error: dynamic type is constexpr-unknown

    constexpr auto [m] = S(1);
    static_assert(m == 1);  // error: lvalue-to-rvalue conversion on mutable
                            // subobject e.m, where e is a
                            // constexpr object of type S

    using A = int[2];
    constexpr auto [v0, v1] = A{2, 3};
    static_assert(v0 + v1 == 5);  // OK, equivalent to e[0] + e[1] where
                                  // e is a constexpr array
}
```

— end example]

[*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 *identifier*s v_0, v_1, v_2, ... 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 *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 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 shall be constant-initializable [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.

[*Example:*

```
struct pixel {
    int x, y;
};
```

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

namespace N {
void f() {
    int x;
    constexpr int& ar = x;          // OK
    static constexpr int& sr = x;   // error: x is not
                                    // constexpr-representable at the point
                                    // indicated below
}
// immediate scope here is that of N
}
```

— end example]

🔲 The `constinit` specifier [dcl.constinit]

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]). [*Note:* A structured binding declaration introduces a uniquely named variable, to which the `constinit` specifier applies. *— end note*] 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.

[*Editor's note:* Add after [temp.arg.nontype]/p6:]

[*Example:*

```
template <int& r> class A{};
extern int x;
A<x> a;  // OK
void f(int p) {
    constexpr int& r = p;  // OK
    A<r> a;                // error: a static constexpr int& variable
                           // cannot be initialized to refer to p here
}
```

— end example]

Acknowledgments

We would like to thanks EDG and Daveed Vandevoorde for providing an implementation.

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, for helping us understand the interaction of this feature with lambda expressions, and for feedback on the wording.

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

Thanks to Daisy Hollman for input on the issue of lambdas that reference variables from their enclosing functions.

Thanks to JF Bastien for encouraging us to clarify the interaction of this feature with lambda expressions.

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. P2280R4: Using unknown references in constant expressions. https://wg21.link/p2280r4, 4 2022.

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

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

[6] 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