constexpr structured bindings

and

references to constexpr variables

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.

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.,

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

which translates to

```cpp
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.,

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

would also simply work as it would be equivalent to

```cpp
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:

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

```c
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:

```c
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:

```cpp
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.

**Next step**

CWG is asking EWG to pick a direction. We will provide wording consistent with that direction. We need to pick one of the options presented in this paper.

- Option 1: Making constexpr implicitly static
- Option 2: Always re-evaluate a call to get
- Option 3: Symbolic addressing

Because Options 1 and 2 can either break existing code or introduce inconsistency, we suggest that the third option, symbolic addressing, constitutes the best path forward.

If we pick that third option, we should further decide whether we want to limit ourselves to allow constexpr references to automatic duration storage objects, or if we should also support constexpr references to thread local duration storage objects, knowing that this will require additional complexity due to the fact that thread-local variables are initialized at the point of use.

In the meantime, this paper retains the wording for constexpr structured binding, which can be pursued independently. (Automatic storage duration bindings of `tuple` will simply not work.) This is the wording for Option 0.

**Wording for constexpr structured binding**

[Editor's note: This wording - corresponding to option 0 - makes constexpr a valid grammatical construct but does not permit automatic storage duration bindings of tuple-like objects.]

[Editor's note: This wording - corresponding to option 0 - makes constexpr a valid grammatical construct but does not permit automatic storage duration bindings of tuple-like objects.]

<table>
<thead>
<tr>
<th>Declarations</th>
<th>[dcl.dcl]</th>
</tr>
</thead>
<tbody>
<tr>
<td>Preamble</td>
<td>[dcl.pre]</td>
</tr>
</tbody>
</table>

[Editor's note: Change 9.1.6 as follow]

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`, `static`, `thread_local`, `auto` [dcl.spec.auto], or a `cv-qualifier`. [Example:

```cpp
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]

A structured binding declaration introduces the identifiers \( v_0, v_1, v_2, \ldots \) 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 constexpr and storage-class-specifiers of the decl-specifier-seq (if any). 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 \( cv1 \) and no ref-qualifier is present, \( e \) is defined by

\[
\text{attribute-specifier-seq opt } S \ cv \ A \ e
\]

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

\[
\text{attribute-specifier-seq opt } \text{decl-specifier-seq ref-qualifier opt } e \ \text{initializer}
\]

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:

\[
\begin{align*}
\text{auto f() -> int(&)[2];} \\
\text{auto [ x, y ] = f();} & \quad \text{// x and y refer to elements in a copy of the array return value} \\
\text{auto& [ xr, yr ] = f();} & \quad \text{// xr and yr refer to elements in the array referred to by f’s return value}
\end{align*}
\]

— 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 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.

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.

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

References


https://wg21.link/N4885