What's the difference between constexpr and const?

Asked 10 years, 8 months ago Modified 21 days ago Viewed 331k times

825

What's the difference between `constexpr` and `const`?

- When can I use only one of them?
- When can I use both and how should I choose one?

c++ c++11 constants constexpr

Share Edit Follow

edited Apr 17, 2022 at 13:59 nbro 15.4k 32 113 197

asked Jan 2, 2013 at 1:42 MBZ 26.1k 47 114 191

131 `constexpr` creates a compile-time constant; `const` simply means that value cannot be changed. – David G Jan 2, 2013 at 1:44

1 Also see When should you use constexpr capability in C++11? – jww Sep 3, 2016 at 3:53

May be this article from `boost/hana` library can enlight some `constexpr` issues where you can use `constexpr` and where you can't: boost.org/doc/libs/1_69_0/libs/hana/doc/html/... – Andry Feb 7, 2019 at 8:38

1 @0x499602D2 "*simply means that value cannot be changed*" For a scalar initialized with a literal, a value that cannot be change *is* also a compile time constant. – curiousguy Jan 20, 2020 at 23:09

@curiousguy Yeah my comment was very oversimplified. Admittedly I was new to `constexpr` back then too :) – David G Jan 20, 2020 at 23:12

10 Answers

Sorted by: Highest score (default)

777

Basic meaning and syntax

Both keywords can be used in the declaration of objects as well as functions. The basic difference when applied to *objects* is this:

- `const` declares an object as *constant*. This implies a guarantee that once initialized, the value of that object won't change, and the compiler can make use of this fact for optimizations. It also helps prevent the programmer from writing code that modifies objects that were not meant to be modified after initialization.
- `constexpr` declares an object as fit for use in what the Standard calls *constant expressions*. But note that `constexpr` is not the only way to do this.

When applied to *functions* the basic difference is this:

- `const` can only be used for non-static member functions, not functions in general. It gives a guarantee that the member function does not modify any of the non-static data members (except for mutable data members, which can be modified anyway).
- `constexpr` can be used with both member and non-member functions, as well as constructors. It declares the function fit for use in *constant expressions*. The compiler will only accept it if the function meets certain criteria (7.1.5/3,4), most importantly [†]:
  - The function body must be non-virtual and extremely simple: Apart from typedefs and static asserts, only a single `return` statement is allowed. In the case of a constructor, only an initialization list, typedefs, and static assert are allowed. (`= default` and `= delete` are allowed, too, though.)
  - As of C++14, the rules are more relaxed, what is allowed since then inside a constexpr function: `asm` declaration, a `goto` statement, a statement with a label other than `case` and `default`, try-block, the definition of a variable of non-literal type, definition of a variable of static or thread storage duration, the definition of a variable for which no initialization is performed.
  - The arguments and the return type must be *literal types* (i.e., generally speaking, very simple types, typically scalars or aggregates)

Constant expressions

As said above, `constexpr` declares both objects as well as functions as fit for use in constant expressions. A constant expression is more than merely constant:

- It can be used in places that require compile-time evaluation, for example, template parameters and array-size specifiers:

```
template<int N>
class fixed_size_list
{ /*...*/ };

fixed_size_list<X> mylist;  // X must be an integer constant expression

int numbers[X];  // X must be an integer constant expression
```

- But note:
- Declaring something as `constexpr` does not necessarily guarantee that it will be evaluated at compile time. It *can be used* for such, but it can be used in other places that are evaluated at run-time, as well.
- An object *may* be fit for use in constant expressions *without* being declared `constexpr`. Example:

```
int main()
{
  const int N = 3;
  int numbers[N] = {1, 2, 3};  // N is constant expression
}
```

This is possible because `N`, being constant and initialized at declaration time with a literal, satisfies the criteria for a constant expression, even if it isn't declared `constexpr`.

So when do I actually have to use `constexpr`?

- An **object** like `N` above can be used as constant expression *without* being declared `constexpr`. This is true for all objects that are:
  - `const` *and*
  - of integral or enumeration type *and*
  - initialized at declaration time with an expression that is itself a constant expression.

[This is due to §5.19/2: A constant expression must not include a subexpression that involves "an lvalue-to-rvalue modification unless [...] a glvalue of integral or enumeration type [...]" Thanks to Richard Smith for correcting my earlier claim that this was true for all literal types.]

- For a **function** to be fit for use in constant expressions, it **must** be explicitly declared `constexpr`; it is not sufficient for it merely to satisfy the criteria for constant-expression functions. Example:

```
template<int N>
class list
{ };

constexpr int sqr1(int arg)
{ return arg * arg; }

int sqr2(int arg)
{ return arg * arg; }

int main()
{
  const int X = 2;
  list<sqr1(X)> mylist1;  // OK: sqr1 is constexpr
  list<sqr2(X)> mylist2;  // wrong: sqr2 is not constexpr
}
```

When can I / should I use both, `const` and `constexpr` *together?*

A. In object declarations. This is never necessary when both keywords refer to the same object to be declared. `constexpr` implies `const`.

```
constexpr const int N = 5;
```

is the same as

```
constexpr int N = 5;
```

However, note that there may be situations when the keywords each refer to different parts of the declaration:

```
static constexpr int N = 3;

int main()
{
  constexpr const int *NP = &N;
}
```

Here, `NP` is declared as an address constant-expression, i.e. a pointer that is itself a constant expression. (This is possible when the address is generated by applying the address operator to a static/global constant expression.) Here, both `constexpr` and `const` are required: `constexpr` always refers to the expression being declared (here `NP`), while `const` refers to `int` (it declares a pointer-to-const). Removing the `const` would render the expression illegal (because (a) a pointer to a non-const object cannot be a constant expression, and (b) `&N` is in-fact a pointer-to-constant).

B. In member function declarations. In C++11, `constexpr` implies `const`, while in C++14 and C++17 that is not the case. A member function declared under C++11 as

```
constexpr void f();
```

needs to be declared as

```
constexpr void f() const;
```

under C++14 in order to still be usable as a `const` function.

Share Edit Follow

edited Aug 21 at 19:13

Benjamin Wang
226 1 9

answered Jan 2, 2013 at 5:10

jogojapan
68.4k 11 101 131

14 IMO the "not necessarily evaluated at compile time" is less helpful than thinking of them as "evaluated at compile time". The constraints on a constant expression mean that it would be relatively easy for a compiler to evaluate it. A compiler must complain if those constraints are not satisfied. Since there are no side effects, you can never tell a difference whether a compiler "evaluated" it or not. – aschepler Jan 2, 2013 at 5:27

21 @aschepler Sure. My main point there is that if you call a `constexpr` function on a non-constant expression, e.g. an ordinary variable, this is perfectly legal and the function will be used like any other function. It will not be evaluated at compile time (because it can't). Perhaps you think that's obvious -- but if I stated that a function declared as `constexpr` will always be evaluated at compile-time, it could be interpreted in the wrong way. – jogojapan Jan 2, 2013 at 5:34

8 Yes, I was talking about `constexpr` objects, not functions. I like to think of `constexpr` on objects as forcing compile time evaluation of values, and `constexpr` on functions as allowing the function to be evaluated at compile time or run time as appropriate. – aschepler Jan 2, 2013 at 5:38

5 A correction: 'const' is only a restriction that YOU cant change the value of a variable; it does not make any promise that the value wont change (ie, by someone else). It's a write property, not a read property. – Jared Grubb Jan 20, 2013 at 17:05

4 This sentence: *It gives a guarantee that the member function does not modify any of the non-static data members.* misses one important detail. Members marked as `mutable` may also be modified by