

constexpr Functions

Chapter 2 Conditionally Safe Features

Employing this cumbersome work-around leads to code that is difficult both to write and to read and is also non-trivial to compile, often resulting in long compile times. What’s more, a separate implementation will be needed for inputs whose values are not **compile-time constants**.

C++11 introduces a new keyword, **constexpr**, that gives users enhanced control over compile-time evaluation. Prepending a function **declaration** with the **constexpr** keyword informs both the compiler and prospective users that the function is eligible for compile-time evaluation and, under the right circumstances, can and will be evaluated *at compile time* to determine the value of a **constant expression**:

```
constexpr int factorial(int n) // can be evaluated in a constant expression
{
    return n == 0 ? 1 : n * factorial(n - 1); // single return statement
}
```

In C++11, the body of a **constexpr** function is effectively restricted to a single **return statement**, and any other language construct, such as **if statements**, loops, **variable** declarations, and so on, are forbidden; see *Restrictions on constexpr function bodies (C++11 only)* on page 268. These seemingly overly strict limitations, although much preferred to the **Factorial metafunction** (e.g., in the code example above), might make optimizing a function’s runtime performance infeasible; see *Potential Pitfalls — Prematurely committing to constexpr* on page 297. As of C++14, however, many of these restrictions were lifted, though some runtime tools remain unavailable during compile-time evaluation. At the time **constexpr** was added to the language, it was a feature under development, and it still is; see Section 2.2. “constexpr Functions ’14” on page 959.

Note that semantic validation of **constexpr** functions occurs only at the point of **definition**. It is therefore possible to **declare** a **member** or **free function** to be **constexpr** for which there can be no valid **definition** — e.g., **constexpr void f()**; — as the return type of a **constexpr** function’s **definition** must satisfy certain requirements, including (in C++11 only) that its return type must not be **void**; see *Restrictions on constexpr function bodies (C++11 only)* on page 268.

Simply declaring a function to be **constexpr** does not automatically mean that the function will necessarily be evaluated at compile time. A **constexpr** function is *guaranteed* to be evaluated at compile time *only* when invoked in a context where a **constant expression** is required.¹ Examples of such contexts include the value of a **non-type template parameter**, array bounds, the first argument to a **static_assert**, **case** labels in **switch statements**, or the initializer for a **constexpr variable**; see Section 2.1. “constexpr Variables” on page 302. If one attempts to invoke a **constexpr** function in a context where a **constant expression** is required with an **argument** that is not a **constant expression**, the compiler will report an error:

¹C++20 formalized this notion with the term **manifestly constant evaluated** to capture all places where the value of an **expression** must be determined at compile time. This new term coalesces descriptions in several places in the Standard where this concept had previously been used without being given a common name.