

Section 2.1 C++11

constexpr Functions

compile time (e.g., the **return** statement in `main`), there is no requirement to have seen the body. In this case, `f9` was not **defined** anywhere within the **translation unit** (TU). Just as with any other **inline** function whose **definition** is never seen, many popular compilers will warn if they see any **expressions** that might invoke such a function, but it is not **ill formed** because the **definition** could, by design, reside in some other TU (see also Section 2.1. “**extern template**” on page 353).

However, when a **constexpr** function is *evaluated* to determine the value of a **constant expression**, its body and anything upon which its body depends must have already been seen; notice that we didn’t say “appears as part of a **constant expression**” but instead said “is evaluated to determine the value of a **constant expression**.”

We *can* have something that is not itself a **constant expression** *appear* as a part of a **constant expression** *provided* that it never actually gets evaluated at compile time:

```
static_assert(true ? true : throw, ""); // OK
static_assert(true ? throw : true, ""); // Error, throw not constexpr

extern bool x;
static_assert((true, x), ""); // Error, x not constexpr
static_assert((x, true), ""); // Error, " " "

static_assert(true || x, ""); // OK
static_assert(x || true, ""); // Error, x not constexpr
```

Note that the *comma* (,) **sequencing operator** incurs evaluation of both of its **arguments**, whereas the *logical-or* (||) **operator** requires only that its two **arguments** be convertible to **bool**, where actual evaluation of the second **argument** might be short circuited.

The type system and function pointers

Similarly to the **inline** keyword, marking a function **constexpr** does *not* affect its **type**; hence, it is not possible to have, say, two **overloads** of a function that differ only on whether they are **constexpr** or to **define** a pointer to exclusively **constexpr** functions:

```
constexpr int f(int) { return 0; } // OK
int f(int)          { return 0; } // Error, int f(int) is now multiply defined.

typedef constexpr int(*MyFnPtr)(int);
// Error, constexpr cannot appear in a typedef declaration.

void g(constexpr int(*MyFnPtr)(int));
// Error, a parameter cannot be declared constexpr.
```

Just as with objects of other types, the value of a function pointer can be read as part of evaluating a **constant expression** only if that pointer is a **compile-time constant**. Furthermore, a function can be invoked at compile time via a function pointer only if the pointer is a **compile-time constant** and the function is declared **constexpr**: