```
static_assert(g(mb),    "");  // Error, mb not usable in a constant expression
static_assert(g(false), "");  // OK
static_assert(g(true),  "");  // OK, j is usable in a constant expression.

int xf = f(mb);  // OK, runtime evaluation of f
int xg = g(mb);  // OK, runtime evaluation of g
```

In the example above, `f` can sometimes be used as part of a constant expression but only if its argument is itself a constant expression and `b` evaluates to **`false`**, thereby avoiding use of the global variable `i`, which is not a compile-time constant. Function `g`, on the other hand, requires only that its argument be a constant expression for it to always be usable as part of a constant expression. If there is not at least one set of compile-time constant argument values that would be usable at compile time, then it is **ill formed, no diagnostic required** (IFNDR):

```
constexpr int h1(bool b) { return f(b); }
    // OK, there is a value of b for which h1 can be evaluated at compile time.

constexpr int h2() { return f(true); }
    // There's no way to invoke h2 so that it can be evaluated at compile time.
    // (This function is ill formed, no diagnostic required.)
```

Here `h1` is **well formed** since it can be evaluated at compile time when the value of `b` is **`false`**; `h2`, on the other hand, is **ill formed** because it can *never* be evaluated at compile time. A sophisticated analysis would, however, be required to establish such a proof, and modern compilers issue a diagnostic only for reasonably simple cases.

Guaranteeing compile-time evaluation for certain arguments is an essential part of a function's **contract**. Declaring a function to be **`constexpr`** might lead prospective clients to conclude that such a function can be evaluated at compile time with *any* compile-time-constant arguments. Such assumptions can prove erroneous as evidenced by `h1` in the example above. Subsequently guaranteeing compile-time evaluation for a wider set of inputs than was originally promised is typically not a problematic change. By contrast, however, providing compile-time evaluation for a narrower set of inputs than was originally available, even if not explicitly promised, can lead to compilation errors for those clients that chose to rely on compile-time usage of the function. It is therefore incumbent on library authors to consider carefully whether to mark a function **`constexpr`** and for which arguments to support compile-time evaluation, since improving the implementation of the function while respecting the restrictions imposed by **`constexpr`** might prove insurmountable, especially with the limitations imposed by C++11; see *Potential Pitfalls — Prematurely committing to **`constexpr`*** on page 297.

Inlining and definition visibility

A function that is declared **`constexpr`** is (1) **implicitly declared `inline`** and (2) automatically eligible for compile-time evaluation. Note that adding the **`inline`** specifier to a function that is already declared **`constexpr`** has no effect: