```
template <typename T>
void f3()
{
    static_assert(sizeof(T) >= 8, "Size < 8.");  // depends on T
}
```

However, see *Potential Pitfalls — Static assertions in templates can trigger unintended compilation failures* on page 120. In the example above, the compiler has no choice but to wait until each time `f3` is instantiated because the truth of the predicate will vary depending on the type provided:

```
void g()
{
    f3<double>();               // OK
    f3<long double>();          // OK
    f3<std::complex<float>>();  // OK
    f3<char>();                 // Error, static assertion failed: Size < 8.
}
```

The Standard does, however, specify that a program containing any template **definition** for which no valid **specialization** exists is **ill formed, no diagnostic required (IFNDR)**, which was the case for `f2` but not `f3` in the example above. Contrast each of the `h*n*` definitions below with its correspondingly numbered `f*n*` definition above[3]:

```
void h1()
{
    int a[!sizeof(int) - 1];  // Error, same as int a[-1];
}

template <typename T>
void h2()
{
    int a[!sizeof(int) - 1];  // Error, always reported
}

template <typename T>
void h3()
{
    int a[!sizeof(T) - 1];    // typically reported only if instantiated
}
```

Both `f1` and `h1` are **ill-formed** nontemplate functions, and both will always be reported at compile time, albeit typically with decidedly different error messages as demonstrated by GCC 10.x's output:

```
f1: error: static assertion failed: Impossible!
h1: error: size -1 of array a is negative
```

[3]The formula used — `int a[-1];` — leads to −1, not 0, to avoid a nonconforming extension to GCC that allows `a[0]`.