Both `f2` and `h2` are ill-formed ~~template functions~~; the cause of their being ill formed ~~has nothing to do with the~~ template ~~type~~ and hence will always be reported as a compile-time error in practice. Finally, `f3` can be only contextually ill formed, whereas `h3` is always necessarily ill formed, and yet neither is reported by typical compilers as such unless and until it has been instantiated. Reliance on a compiler not to notice that a program is ill formed is dubious; see *Potential Pitfalls — Static assertions in templates can trigger unintended compilation failures* on page 120.

Use Cases

Verifying assumptions about the target platform

Some programs rely on specific properties of the native types provided by their target platform. Static assertions can help ensure portability and prevent such programs from being compiled into a malfunctioning binary on an unsupported platform. As an example, consider a program that relies on the size of an **int** to be exactly 32 bits, e.g., due to the use of inline **asm** blocks. Placing a **static_assert** in namespace scope in any of the program's translation units will ensure that the assumption regarding the size of **int** is valid, and also serve as documentation for readers:

```
#include <climits>  // CHAR_BIT

static_assert(sizeof(int) * CHAR_BIT == 32,
    "An int must have exactly 32 bits for this program to work correctly.");
```

More typically, statically asserting the *size* of an **int** avoids having to write code to handle an **int** type's having greater or fewer bytes when no such platforms are likely ever to materialize:

```
static_assert(sizeof(int) == 4, "An int must have exactly 4 bytes.");
```

Preventing misuse of class and function templates

Static assertions are often used in practice to constrain class or function templates to prevent their being instantiated with unsupported types. If a type is not syntactically compatible with the template, static assertions provide clear customized error messages that replace compiler-issued diagnostics, which are often absurdly long and notoriously hard to read. More critically, static assertions actively avoid erroneous runtime behavior.

As an example, consider the `SmallObjectBuffer<N>` class templates,[4] which provide storage, aligned properly using **alignas** (see Section 2.1."**alignas**" on page 168), for arbitrary objects whose size does not exceed `N`:

[4] A `SmallObjectBuffer` is similar to C++17's `std::any` (**cppprefc**) in that it can store any object of any type. Instead of performing dynamic allocation to support arbitrarily sized objects, however, `SmallObjectBuffer` uses an internal fixed-size buffer, which can lead to better performance and cache locality provided the maximum size of all of the types involved is known.