```
// C++03 (obsolete)
#include <cassert>  // standard C assert macro

template <typename T>
struct Pi {
    static const T value;
};

template <typename T>
const T Pi<T>::value(3.1415926535897932385);  // separate definition

void testCpp03Pi()
{
    const float       piAsFloat      = 3.1415927;
    const double      piAsDouble     = 3.141592653589793;
    const long double piAsLongDouble = 3.1415926535897932385;

    // additional boilerplate on use (::value)
    assert(Pi<float>::value       == piAsFloat);
    assert(Pi<double>::value      == piAsDouble);
    assert(Pi<long double>::value == piAsLongDouble);
}
```

Use Cases

Parameterized constants

A common effective use of variable templates is in the definition of type-parameterized constants. As discussed in *Description* on page 157, the mathematical constant π serves as our example. Here we want to initialize the constant as part of the variable template; the literal chosen is the shortest decimal string to do so accurately for an 80-bit **`long double`**:

```
template <typename T>
constexpr T pi(3.1415926535897932385);
    // smallest digit sequence to accurately represent pi as a long double
```

For portability, a floating-point literal value of π that provides sufficient precision for the longest **`long double`** on any relevant platform — e.g., 34 decimal digits for 128 bits of precision: `3.141'592'653'589'793'238'462'643'383'279'503` — might be used; see Section 1.2.“Digit Separators” on page 152.

Notice that we have elected to use **`constexpr`** variables in place of **`const`** to guarantee that the floating-point `pi` is a compile-time constant that will be usable as part of a constant expression.