Note that being marked **`constexpr`** enables a function to be evaluated *at compile time* only if (1) the argument values are constant expressions known before the function is evaluated and (2) no operations performed when invoking the function with those arguments involve any of the excluded ones listed above.

Global variables can be used in a **`constexpr`** function only if they are (1) *non*-**`volatile`** **`const`** objects of *integral* or *enumerated* type that are initialized by a constant expression (generally treated as **`constexpr`** even if only marked as **`const`**), or (2) **`constexpr`** objects of literal type; see *Literal types defined* on page 278 and Section 2.1."**`constexpr`** Variables" on page 302. In either case, any **`constexpr`** global object used within a **`constexpr`** function must be initialized with a constant expression prior to the definition of the function. C++14[15] relaxes some of these restrictions; see Section 2.2."**`constexpr`** Functions '14" on page 959.

Use Cases

A better alternative to function-like macros

Computations that are useful both at run time and at compile time and/or that must be inlined for performance reasons were typically implemented using preprocessor macros. For instance, consider the task of converting mebibytes to bytes:

```
#define MEBIBYTES_TO_BYTES(mebibytes) ((mebibytes) * 1024 * 1024)
```

The macro above can be used in contexts where both a constant expression is required and the input is known only during program execution:

```
#include <cstddef>  // std::size_t
#include <vector>   // std::vector
void example0(std::size_t input)
{
    unsigned char fixedBuffer[MEBIBYTES_TO_BYTES(2)];  // compile-time constant

    std::vector<unsigned char> dynamicBuffer;
    dynamicBuffer.resize(MEBIBYTES_TO_BYTES(input));  // usable at run time
}
```

While a single-line macro with a reasonably unique (and long) name like `MEBIBYTES_TO_BYTES` is unlikely to cause any problems in practice, it harbors all the disadvantages macros have compared to regular functions. Macro names are not scoped; hence, they are subject to global name collisions. There is no well-defined input and output type and thus no type safety. Perhaps most tellingly, the lack of expression safety makes writing even simple macros tricky; a common error, for example, is to forget the `()` around `mebibytes` in the implementation of `MEBIBYTES_TO_BYTES`, resulting in an unintended result if applied to a non-trivial expression such as `MEBIBYTES_TO_BYTES(2+2)` — yielding a value of `(2+2 * 1024 * 1024) = 2097154` without the `()` and the intended value of `((2+2) * 1024 * 1024) = 4194304` with them.

[15] C++17 and C++20 each further relax these restrictions.