```
template <typename T>
constexpr int badSizeOf(T t) { const int s = sizeof(t); return s; }
    // This constexpr function template is IFNDR.
```

~~Most~~ compilers, when compiling such a specialization for runtime use, will not attempt to determine if the **constexpr** would ever be valid. When invoked with arguments that are themselves constant expressions, they do, however, often detect this ill-formed nature and report the error:

```
int d[badSizeOf(S1())];  // Error, badSizeOf<S1>(S1) body not return statement
int e[badSizeOf(S0())];  // Error, badSizeOf<S0>(S0) body not return statement
int f = badSizeOf(S1()); // Oops, same issue but might work on some compilers
int g = badSizeOf(S0()); // Oops, same issue but often works without warnings
```

Importantly, note that each of the four statements in the code snippet above is ill formed because the `badSizeOf` function template is itself ill formed. Although the compiler is not required to diagnose the general case, it is ill formed to attempt to use an instantiation of `badSizeOf` in a context requiring a constant expression, e.g., `d` or `e`. When used in a context not requiring a constant expression (e.g., `f` or `g`), whether the compiler fails, warns, or proceeds is a matter of **quality of implementation (QoI)**.

constexpr-function parameter and return types

At this point, we arrive at what is perhaps the most confounding part of the seemingly *cyclical* definition of **constexpr** functions: A function cannot be declared **constexpr** unless the return type and every parameter of that function satisfies the criteria for being a literal type, i.e., the category of types whose objects are permitted to be created and destroyed when evaluating a constant expression:

```
struct  Lt { int v; constexpr  Lt() : v(0) { } };  // literal type
struct Nlt { int v;           Nlt() : v(0) { } };  // nonliteral type

          Lt  f1() { return Lt();  }  // OK, no issues
constexpr Lt  f2() { return Lt();  }  // OK, returning literal type
          Nlt f3() { return Nlt(); }  // Ok, function is nonconstexpr.
constexpr Nlt f4() { return Nlt(); }  // Error, constexpr returning nonliteral

          int g1(Lt  x) { return x.v; }  // OK, no issues
constexpr int g2(Lt  x) { return x.v; }  // OK, parameter is a literal type.
          int g3(Nlt x) { return x.v; }  // OK, function is nonconstexpr.
constexpr int g4(Nlt x) { return x.v; }  // Error, constexpr taking nonliteral
```

Consider that all *pointer* and *reference* types — being *built-in types* — are literal types and therefore can appear in the interface of a **constexpr** function irrespective of whether they point to a literal type: