No static constexpr data members defined in their own class

When implementing a class using the singleton pattern, it might be desirable to have the single object of that type be a **constexpr private static** member of the class itself, with guaranteed compile-time, **data-race**-free initialization and no direct accessibility outside the class. This approach does not work as easily as planned because **constexpr static** data members must have a **complete type**, and the class being defined is not complete until its closing brace:

```
class S
{
private:
    static const     S constVal;      // OK, initialized outside class below
    static constexpr S constexprVal;  // Error, constexpr must be initialized.
    static constexpr S constInit{};   // Error, S is not complete.
};

const S S::constVal{};  // OK, initialize static const member.
```

The "obvious" workaround of applying a more traditional singleton pattern, where the singleton object is a static local variable at **function scope**, also fails (see Section 1.1. "Function **static** '11" on page 68) because **constexpr** functions are not allowed to have static variables (see Section 2.1."**constexpr** Functions" on page 257):

```
constexpr const S& singleton()
{
   static constexpr S object{};  // Error, even in C++14, static is not allowed.
   return object;
}
```

The only solution available for **constexpr** objects of **static storage duration** is to put them outside of their type, either at global scope, at **namespace** scope, or nested within a befriended helper class[8]:

```
class S
{
    friend struct T;
    S() = default;  // private
    // ...
};

struct T
{
    static constexpr S constInit{};
};
```

[8] ~~C++20 provides an alternate partial solution with the **constinit** keyword, allowing for compile-time initialization of static data members, but that still does not make such objects usable in a constant expression.~~