```
struct Metrics
{
    int d_numRequests;  // number of requests to the service
    int d_numErrors;    // number of error responses

    Metrics(int, int);  // user-provided value constructor

    // Generation of default constructor is suppressed.
};
```

As illustrated in *Appendix — Implicit Generation of Special Member Functions* on page 44, the presence of a user-provided constructor suppressed the implicit generation of the default constructor. Replacing the default constructor with a seemingly equivalent user-provided one might appear to work as intended:

```
struct Metrics
{
    int d_numRequests;  // number of requests to the service
    int d_numErrors;    // number of error responses

    Metrics(int, int);  // user-provided value constructor
    Metrics() {}        // user-provided default constructor

    // Default constructor is user-provided: Metrics is not trivial.
};
```

The user-provided nature of the default constructor, however, renders the `Metrics` type non-trivial, even ~~if~~ the ~~definitions are identical~~! In contrast, explicitly requesting the default constructor be generated using `= default` restores the triviality of the type:

```
struct Metrics
{
    int d_numRequests;  // number of requests to the service
    int d_numErrors;    // number of error responses

    Metrics(int, int);    // user-provided value constructor
    Metrics() = default;  // defaulted, trivial default constructor

    // Default constructor is defaulted: Metrics is trivial.
};
```

Physically decoupling the interface from the implementation

Sometimes, especially during large-scale development, avoiding compile-time coupling clients to the implementations of individual methods offers distinct maintenance advantages.