Potential Pitfalls

Loss of insulation

Although convenient, placing default values in a header file — and thus potentially also using a **default** default constructor — can result in a loss of insulation that can have severe consequences, especially at scale. For instance, consider a hash table with a non**static** data member representing the growth factor:

```
// hashtable.h:

class HashTable
{
private:
    float d_growthFactor;
    // ...

public:
    HashTable();
    // ...
};
```

Without using default member initializers, the default growth factor is provided as part of the member initializer list of the default constructor:

```
// hashtable.cpp:
#include <hashtable.h>  // HashTable

HashTable::HashTable() : d_growthFactor(2.0f) { }
```

In the eventuality that the default growth factor is too large and results in excessive memory consumption in production, relinking the affected applications with a new version of the library-provided `HashTable`, rather than recompiling them, is sufficient. Subject to a company's compilation and deployment infrastructure, relinking alone can be significantly less expensive than having to recompile the entire program prior to relinking it.

Had the default member initializer been used, the otherwise trivial default constructor might be defined in the header with `= default`, effectively removing any insulation of these values that might allow speedy relinking in lieu of expensive recompilation, should these values need to change in a crisis.[2]

Inconsistent subobject initialization

An approach occasionally taken to avoid keeping globally shared state is to have objects keep a handle to a `Context` object holding data that would otherwise be application-global:

```
struct Context
{
```

[2] For a complete description of this real-world example, see **lakos20**, section 3.10.5, pp. 783–789.