When Singletons go bad..

Richard Shepherd

Simple thread safe construction for a shared resource

`static` object is only constructed once

```
class Manager1
{
public:
  static Manager1& Instance() {
    static Manager1 manager;
    return manager;
  }
  std::string const * GetResource() const {
```

Uncertain initialization?

```
class Manager3
{
  Resource * resource_;
  Manager3(): resource_{CreateResource()}{}
  ~Manager3() {
    if (resource_)
      DestroyResource(resource_);
  }
public:
  static Manager3& Instance() {
    static Manager3 manager;
    return manager;
  }
  Resource const * GetResource() const {
```

Only one chance for construction

```
if (const auto res = Manager3::Instance().GetResource())
  DoWork(res);
if (const auto res = Manager3::Instance().GetResource())
  DoWork(res);
```

Fixes

- for each use: check `IsValid` and `ReInit`
- repeated creation with manual flag and mutex
- some more complex static which included constructing another static capturing the resource
- nothing that had the reliability, portability or elegance of static

Re-examine static construction

`static` object only **completes** construction once.

.. and throwing an exception interrupts construction...

```
class Manager4
{
  Manager4(): resource_{CreateResource()} {
    if (!resource_)
      throw std::exception{"Not ready"};
  }
public:
  static Manager4* Instance() {
    try {
      static Manager4 manager;
      return &manager;
    }
    catch(...) {
      return nullptr;
    }
```

API encourages checking singleton's readiness

```
if (const auto mgr = Manager4::Instance())
  DoWork(mgr->GetResource());
if (const auto mgr = Manager4::Instance())
  DoWork(mgr->GetResource());
```

Takeaways

- Successful construction of an object doesn't mean it's useable
- Don't forget exceptions as part of the constructor/destructor toolkit
-