Ref (Pointers & Base)

December 2025

Ref – Overview

- Ref – Pointer
  - Points to anything derived from Refable
  - Copy it around
  - Implicit validity check
- Refable – Expected base
  - inc/dec ref methods
  - Virtual destructor
- RefCount – Usual base
  - Derives from Refable

```
struct MyClass : public RefCount {};

Ref<MyClass> ref = new MyClass;

if(ref){

    // do stuff, pointer is valid

}

MyClass* ptr = ref.ptr();

ref = {}; // decrements reference, which
          // RefCount deletes if zero
```

Ref – Tradeoffs

- Ref Advantages
  - Ref is one pointer
  - Can create valid ref from pointer w/o losing count
  - Destructors can be protected/private
    - Make Ref a friend
  - Control the zero condition with custom refable base
    - Caching
  - One line forward declaration footprint
- Ref Disadvantages
  - Count is internal
  - To have POD, must use either inheritance or encapsulation

- Shared Ptr Advantages
  - Count is external
  - Can treat data as POD

- Shared Ptr Disadvantages
  - Two pointers inside (data & control)
  - Destructor **MUST** be public
  - Must include header to forward declare
  - Zero count **IS** destruction w/o using the secondary “default” template argument (and count on one hand how many coders are expecting *that*).

Ref – Thread Safety

- Refable Based
  - Counts use std::atomic
    - Inherently threadsafe
  - Derived class data; usual caveats apply.
    - Guaranteed read-only is fine
    - Read/Write data… requires mutexes or similar synchronization

- Ref Class
  - Instance itself (ie, a global variable Ref<MyClass> gMyClass)
  - Usual caveats apply to the instance itself; read only is fine, read/write requires synchronization considerations.

Ref – Refable

- Refable
  - Preferred root class
  - Helps metaprogramming
- RefCount
  - Usual Base class
  - Deletes on decrement to zero
- RefQ
  - For Qt's Qobject
  - Calls "deleteLater()" on zero decrement
- Resource
  - For resources
  - Makes eligible for cache removal on zero decrement