UCONN

The Pointers Strike Back

Laurent Michel
Computer Science & Engineering

Outline

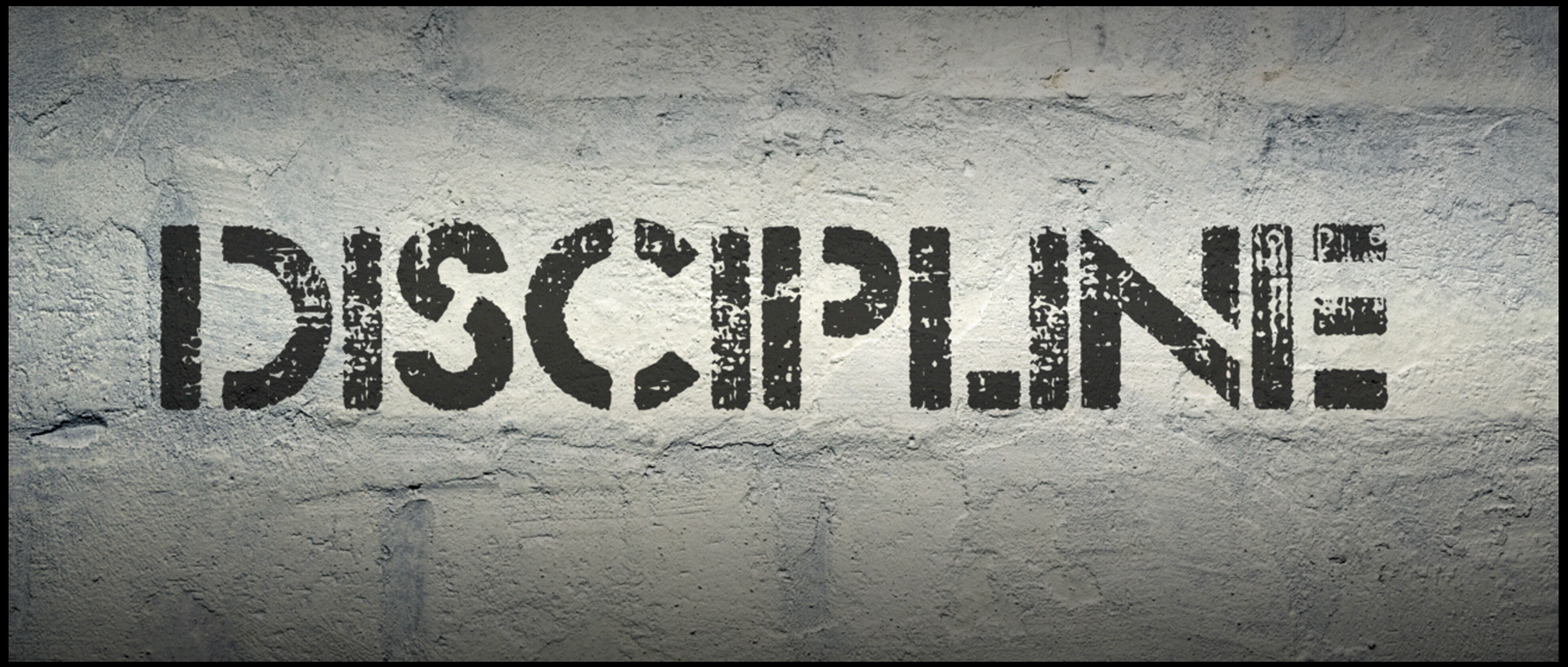
- C and RAW pointers
- C++ unique pointers
- C++ shared pointers
- C++ weak pointers

RAW Pointers

- Straight from C
  - You are in charge of **everything**
    - **Allocating**
    - **Deallocating**
- Typical challenge
    - Who should deallocate something and when ?

UCONN

The C Way



The C++ Way

- Introduce new / additional concepts
  - Each one enforces a type of “best practice”
- Requirement
  - Be systematic.
  - Don’t “cheat” your way back out to RAW pointers

Concept

C++

Concept #1

The Strict Ownership Doctrine

`unique_ptr<T>`

http://en.cppreference.com/w/cpp/memory/unique_ptr

Documentation

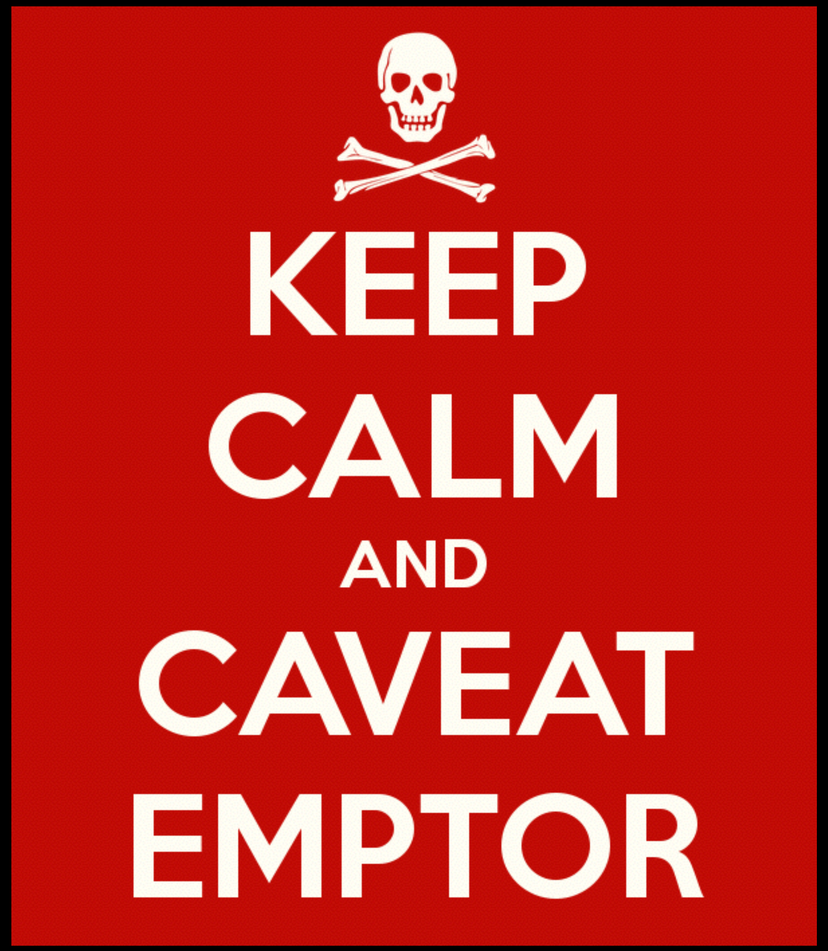
std::unique_ptr is a **smart pointer** that retains **sole ownership** of an object through a pointer and **destroys that object when the unique_ptr goes out of scope**. No two unique_ptr instances can manage the same object.

Ownership Transfer

- Doctrine
  - Only **one** owner
- Assignments **transfer** ownership by explicitly ***moving*** *the pointer*
- Explicit move applies to
  - Normal assignments
  - Argument passing
- Deallocation is automatic on unique_ptr death.

To get or not to get ?

- The “get” method should not be used…
- It only exists for legacy reasons
  - Passing a unique to something that does not conform to the unique_ptr<T> API.



Demo Time

Concept

C++

Concept #2

The Shared Ownership Doctrine

`shared_ptr<T>`

http://en.cppreference.com/w/cpp/memory/shared_ptr

Documentation

std::shared_ptr is a smart pointer that retains **shared ownership** of an object through a pointer. **Several shared_ptr objects may own the same object.** The object is destroyed and its memory **deallocated when either of the following happens:**

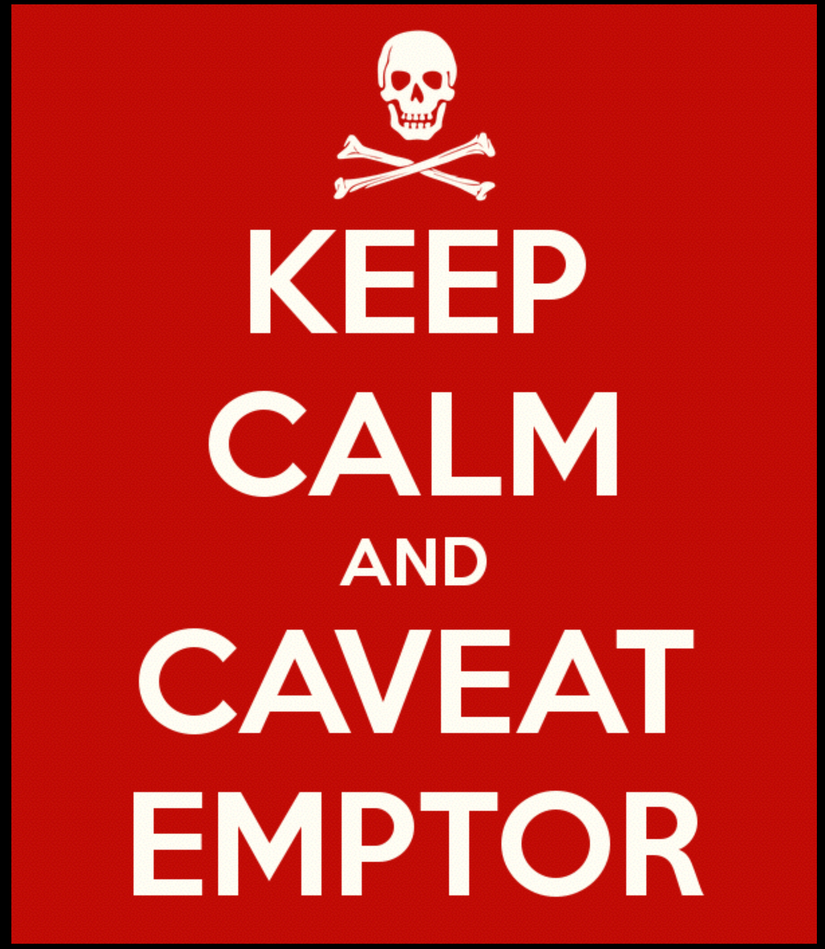
- the last remaining shared_ptr owning the object is destroyed;
- the last remaining shared_ptr owning the object is assigned another pointer via operator= or reset()

Illustrations

- Creation
- Sharing
- Destruction
- Assignments

To get or not to get ?

- The “get” method should not be used…
- It only exists for legacy reasons
  - Passing a shared to something that does not conform to the shared_ptr<T> API.
- use_count API
  - Useful to figure out “how many refs!”



Demo Time

Sharing “This”

- Consider a simple example

```
#include <iostream>
#include <memory>

struct Foo {
   int a;
   Foo(int x) { a = x;}
   ~Foo()     { std::cout << "Destroying Foo(" << this << ")" << std::endl;}
   std::shared_ptr<Foo> getThis() { return std::shared_ptr<Foo>(this);}
};
```

The solution

- Use a special class

std::enable_shared_from_this

- And adopt a “can” implementation

```
#include <iostream>
#include <memory>

struct Foo : std::enable_shared_from_this<Foo> {
    int a;
    Foo(int x) { a = x;}
    ~Foo()     { std::cout << "Destroying Foo(" << this << ")" << std::endl;}
    std::shared_ptr<Foo> getThis() { return shared_from_this();}
};
```

Concept #3

The Weak Ownership Doctrine

`weak_ptr<T>`

http://en.cppreference.com/w/cpp/memory/weak_ptr

Documentation

std::weak_ptr is a smart pointer that holds a **non-owning ("weak") reference** to an object that is managed by std::shared_ptr. **It must be converted to std::shared_ptr in order to access** the referenced object.

Illustration

- Creation
- Conversion (lock)

Demo Time