# C++11 SMART POINTERS

@mine260309

# AGENDA

- Overview
- shared_ptr
- unique_ptr
- weak_ptr
  - Cyclic reference problem
  - Enable shared from this
- Miscs

# OVERVIEW

- std::shared_ptr
  - shared ownership
- std::unique_ptr
  - unique ownership
- std::weak_ptr
  - No ownership
- NO std::intrusive_ptr

≈

- boost::shared_ptr
  - shared ownership
- boost::scoped_ptr
  - unique ownership
- boost::weak_ptr
  - No ownership

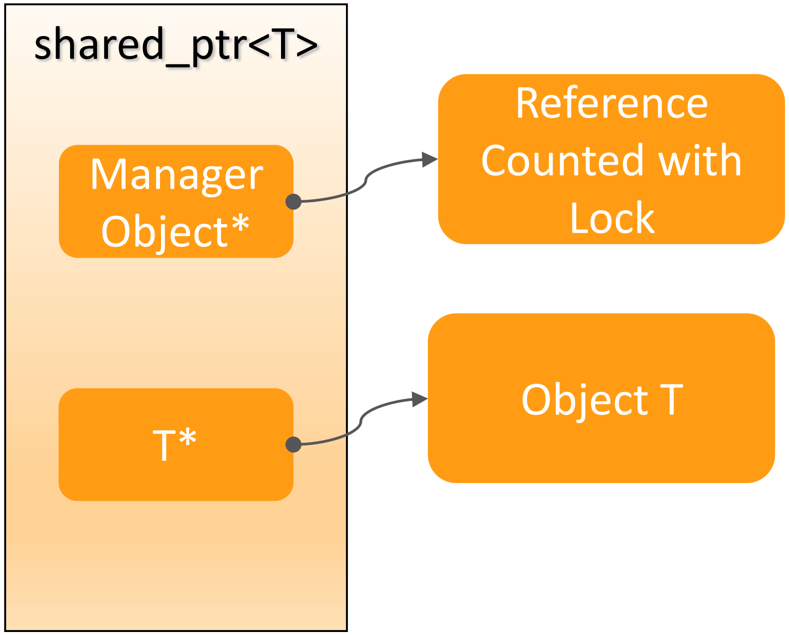
Differences:

- Compiler (C++11 vs C++03)
- Move-Semantic
  - *std::unique_ptr* supports transfer-of-ownership
  - *boost::scoped_ptr* is neither copyable nor movable
- Array support (See details in Miscs)

# std::shared_ptr

# SHARED_PTR

- Features
  - share ownership
  - reference count
  - auto delete
  - native inheritance
  - cast



- Overhead
  - Manager-Object*
  - Managed-Object-T*
  - Lock of increment/decrement of reference count (Thread safe)

# RULES OF SHARED_PTR

- Refer to objects allocated with ***new*** and can be deleted with ***delete***

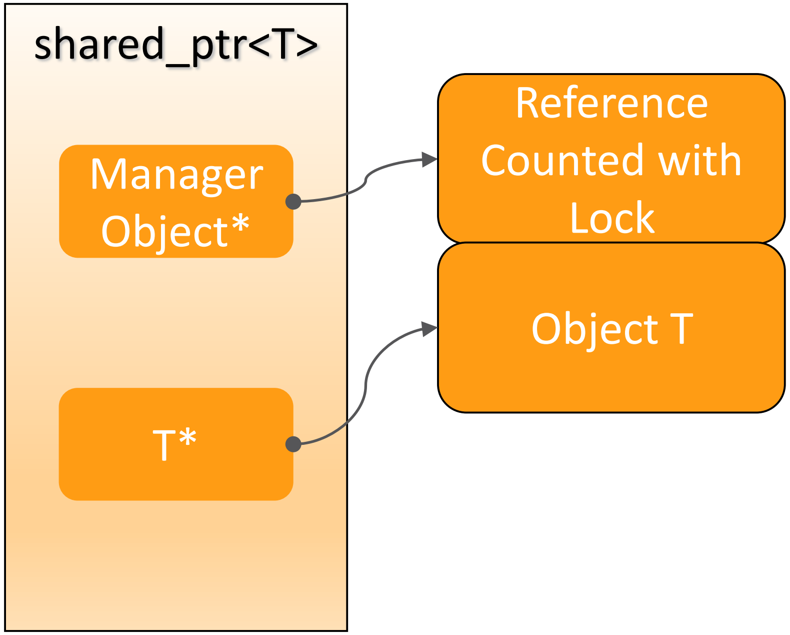
- Create by ***new*** or ***make_shared***
  - `shared_ptr<T> t(new T(…));`
  - `shared_ptr<T> t(make_shared<T>(…));`

- Try **hard** to avoid using raw pointers
  - Mixing smart and built-in pointers can be hard to get right

- Then never explicitly call ***delete***

# MAKE_SHARED VS NEW

- `shared_ptr<T> t(new T(…));`
  - Two dynamic allocations
- `shared_ptr<T> t(make_shared<T>(…));`
  - Single dynamic allocation



- Why?
  - Manager-Object*
  - Managed-Object-T*

- Prefer ::make_shared if a lof of shared_ptrs are created

# BASIC USE OF SHARED_PTR

- http://ideone.com/aEFxlk

```
std::shared_ptr<TClass> c2(new TClass(2)); // TClass 2
std::shared_ptr<TClass> c3 = std::make_shared<TClass>(3);
std::shared_ptr<TClass> c4; // Empty shared_ptr
c4.reset(new TClass(4)); // TClass: 4
if (c4) {
  ... // Do something
}
c4.reset(); // c4 becomes empty
if (c4) { // Now it returns false
  ... // Code does not go here
}
```

# INHERITANCE OF SHARED_PTR

- Same as raw pointer
- http://ideone.com/jp4iCI

```
std::shared_ptr<TDerived> dp1(new TDerived);
std::shared_ptr<TBase> bp1 = dp1;
std::shared_ptr<TBase> bp2(dp1);
std::shared_ptr<TBase> bp3(new TDerived);
```

# CASTING SHARED_PTR

- Similar with raw pointer
  - *static_pointer_cast*
  - *dynamic_pointer_cast*
  - *const_pointer_cast*
- Create a new shared_ptr!
- http://ideone.com/TdcPDl

```
std::shared_ptr<TBase> bp1(new TDerived);
std::shared_ptr<const TBase> cbp(new TBase);

std::shared_ptr<TDerived> dp1 = std::static_pointer_cast<TDerived>(bp1);
std::shared_ptr<TDerived> dp2 = std::dynamic_pointer_cast<TDerived>(bp1);
std::shared_ptr<TBase> bp2 = std::const_pointer_cast<TBase>(cbp);
//std::shared_ptr<TDerived> d = static_cast<std::shared_ptr<TDerived>>(bp1);
// Compile error
```

# std::unique_ptr

# UNIQUE_PTR

- Features
  - Unique ownership
    - Copy constructor and copy assignment = delete
  - No reference count
  - auto delete
  - native inheritance
  - **No** cast, or **manually** cast
- Overhead
  - Nothing!
- Rules?
  - The same as *shared_ptr*

# BASIC USE OF UNIQUE_PTR

- `new` or `std::move` (transfer ownership)
- http://ideone.com/bxsFvC

```
std::unique_ptr<TClass> c2(new TClass(2));
std::unique_ptr<TClass> c3; // Empty unique_ptr
//c3 = c2; // error: use of deleted function operator=()
c3 = std::move(c2); // unique_ptr has to be moved
                    // Now c2 owns nothing

// Note that return value of a function is a rvalue
std::unique_ptr<TClass> GetATClass() {
  std::unique_ptr<TClass> c(new TClass(0));
  return c; // same as `return std::move(c);`
}
c3 = GetATClass();
```

# INHERITANCE OF UNIQUE_PTR

- Same as raw pointer
- http://ideone.com/FhgRi9

```
std::unique_ptr<TDerived> dp1(new TDerived);
std::unique_ptr<TBase> bp1 = std::move(dp1);
std::unique_ptr<TBase> bp2(std::move(bp1));
std::unique_ptr<TBase> bp3(new TDerived);
```

# CAST(MANUALLY) OF UNIQUE_PTR

- Generally, do **NOT** cast
- Why no native cast?
  - Cast makes a copy of the pointer
- But I do want to cast unique_ptr?
- http://ideone.com/F8CfIG

```
std::unique_ptr<TBase> bp1(new TDerived);

std::unique_ptr<TDerived> dp1(static_cast<TDerived*>(bp1.get()));
bp1.release(); // Now bp1 owns nothing

bp1 = std::move(dp1); // Transfer ownership to bp1 (inheritance)

std::unique_ptr<TDerived> dp2(dynamic_cast<TDerived*>(bp1.get()));
bp1.release(); // Now bp1 owns nothing
```

# std::weak_ptr

# WEAK_PTR

- “Observe” the managed object
- Provide a *shared_ptr* when used

- Why?
  - Solve cyclic reference of *shared_ptr*
  - Helps to get a *shared_ptr* from “this”

# BASIC USE OF WEAK_PTR

- http://ideone.com/tZ3ZhJ

```
std::weak_ptr<TClass> w; // Empty weak_ptr
{
  std::shared_ptr<TClass> c(new TClass); // TClass: -1
  std::weak_ptr<TClass> w1(c); // Construct from shared_ptr
  std::weak_ptr<TClass> w; // Empty weak_ptr
  w = c;
  std::weak_ptr<TClass> w3(w);
  w3.reset(); // w3 becomes empty
  w3 = w; // w3 points to the TClass as well
  std::shared_ptr<TClass> c2 = w.lock(); //Get shared_ptr by weak_ptr
  c2->IntValue = 1;
} // ~TClass: 1
std::shared_ptr<TClass> c3 = w.lock(); // c3 is empty shared_ptr
```

# CYCLIC REFERENCE PROBLEM

- http://ideone.com/KP8oSL

```
class CyclicA {
public:
  shared_ptr<CyclicB> b;
};
class CyclicB {
public:
  shared_ptr<CyclicA> a;
};
void TestSharedPtrCyclicRef()
{
  shared_ptr<CyclicA> a(new CyclicA);
  shared_ptr<CyclicB> b(new CyclicB);
  a->b = b;
  b->a = a;
} // Neither a nor b is deleted
```

# CYCLIC REFERENCE - FIX

- http://ideone.com/KP8oSL

```
class FixCyclicA {
public:
  std::shared_ptr<FixCyclicB> b;
};
class FixCyclicB {
public:
  std::weak_ptr<FixCyclicA> a;
};
void TestWeakPtrFixCyclicRef()
{
  std::shared_ptr<FixCyclicA> a(new FixCyclicA);
  std::shared_ptr<FixCyclicB> b(new FixCyclicB);
  a->b = b;
  b->a = a;
} // Both a and b are deleted
```

# ENABLE SHARED FROM THIS - WHY

- How to get `shared_ptr` from class’s member function?

```
class TShareClass {
  ...
  std::shared_ptr<TShareClass> GetThis() {
    // how to achieve?
  }
  void CallFoo() {
    Foo(GetThis());
  }
}
void Foo(const std::shared_ptr<TShareClass>& s)
{
  // Do something to s, e.g. s->xxx = xxx
}
```

# ENABLE SHARED FROM THIS – THE WRONG WAY

- A wrong way

```
class TShareClass {
  ...
  std::shared_ptr<TShareClass> GetThis () {
    return std::shared_ptr<TShareClass>(this);
  } // This gets deleted after out-of-scope
}
{
  std::shared_ptr<TShareClass> a(new TShareClass);
  std::shared_ptr<TShareClass> temp = a.GetThis();
} // Deleted twice!
```

# ENABLE SHARED FROM THIS – AN ATTEMP

- One way to achieve: Add a weak_ptr

```
class TMyShareClass
{
public:
  std::shared_ptr<TMyShareClass> GetThis() {
    return MyWeakPtr.lock(); // Make sure MyWeakPtr is valid
  }
  std::weak_ptr<TMyShareClass> MyWeakPtr;
};

std::shared_ptr<TMyShareClass> c1(new TMyShareClass());
c1->MyWeakPtr = c1;
std::shared_ptr<TMyShareClass> c2 = c1->GetThis();
```

# ENABLE SHARED FROM THIS – A DECENT WAY

- C++11's built-in *enable_shared_from_this*
- http://ideone.com/wRUj3U

```
class TShareClass : public std::enable_shared_from_this<TShareClass>
{
  ...
  std::shared_ptr<TShareClass> GetThis() {
    return shared_from_this();
  }
};
std::shared_ptr<TShareClass> c1(new TShareClass());
std::shared_ptr<TShareClass> c2 = c1->GetThis();
```

# ENABLE SHARED FROM THIS – BE CAREFUL

- Do not call `shared_from_this()` from constructor
  - *weak_ptr* is not valid yet in ctor
- Always create shared_ptr<T>, never create raw T*

```
TShareClass* c1 = new TShareClass();
std::shared_ptr<TShareClass> c2 = c1->GetThis();
  // Undefined behavior
  // Throws exception 'std::bad_weak_ptr` on gcc 4.9.x
```

- Consider make ctor/copy-ctors private and unique the creation
  - Prevent creating raw T in case of wrong usage
  - Benefit from perfect forwarding

# ENABLE SHARED FROM THIS – BEST PRACTICE

- Perfect creation of T (http://ideone.com/UyIPgb)

```
class TPerfectCtor : public std::enable_shared_from_this<TPerfectCtor>
{
private:
  TPerfectCtor(int I = -1) = default;
  TPerfectCtor(const TPerfectCtor& r) = default;
public:
  template<typename ... T>
  static std::shared_ptr<TPerfectCtor> Create(T&& ... all) {
    return std::shared_ptr<TPerfectCtor>(
      new TPerfectCtor(std::forward<T>(all)...));
  }
  std::shared_ptr<TPerfectCtor> GetThis() {
    return shared_from_this();
  }
};
// std::shared_ptr<TPerfectCtor> c1(new TPerfectCtor()); // compile error
std::shared_ptr<TPerfectCtor> c1 = TPerfectCtor::Create(); // TPerfectCtor: -1
std::shared_ptr<TPerfectCtor> c2 = TPerfectCtor::Create(2); // TPerfectCtor: 2
c2 = c1->GetThis(); // ~TPerfectCtor: 2
```

# Miscs

# MISCS

- Default, use *unique_ptr*
- Default, use *unique_ptr* in containers
  - `std::vector<std::unique_ptr<T>>`
- If the object has shared ownership, use *shared_ptr*
- If the objects have shared ownership, use *shared_ptr* in containers
  - `std::vector<std::shared_ptr<T>>`
- Prefer to pass by const reference
  - `void Foo(const std::shared_ptr<T>& sp);`
  - `void Foo(const std::unique_ptr<T>& up);`

  Do not write like below

  - `void Foo(std::shared_ptr<T>& sp); // Sometimes compile error`
  - Why? *sp.reset(new Base)* while sp is *Derived*

# MISCS – ARRAY SUPPORT

```
std::unique_ptr<T[]> ua(new T [5]); // OK
boost::scoped_ptr<T[]> ua(new T [5]); // Compile error

std::shared_ptr<T[]> ua(new T [5]); // Compile error
boost::shared_ptr<T []> a(new T [5]); // OK (since Boost 1.53)

// A custom deleter for array
std::shared_ptr<T> a(new T [5], std::default_delete<T[]>());
// OK, but access with a.get()[index]

// Never pass T[] to shared_ptr<T>
std::shared_ptr<T> a(new T [5]); // Crash
boost::shared_ptr<T> a(new T [5]); // Crash
```

# MISCS – CONT.

- Suggested ways to use array in smart pointer
  - `std::unique_ptr<T[]>`
  - `std::shared_ptr<T> with custom delete`
  - `boost::shared_ptr<T[]> (Since Boost 1.53)`
  - `boost::shared_array<T>`

- Consider `boost::ptr_vector<T>` for vector of shared_ptr if performance is critical

- http://ideone.com/n9lZJ2

# FURTHER READINGS

- boost's  Pointer Container Library
  - ptr_sequence_adapter
    - ptr_vector
    - ptr_list
    - ptr_deque
    - …
  - associative_ptr_container
    - ptr_set_adapter
    - ptr_multiset_adapter
    - ptr_map_adapter
    - …
- boost::scoped_array
- boost::intrusive_ptr

# Thank You!