Core Guidelines - Smart Pointers

Yossi Moalem

Smart pointers

- Wrapper class
- Behaves like raw (bare) pointer
  - Overloads operator ->, operator *, etc.
- Automatic memory management

```
std::unique_ptr<Foo> foo(new Foo);
foo->doSomething();
//no need to call delete
```

Ownership

```
Foo * foo = new Foo;
Bar(foo);
```

Does bar takes ownership?

Should we release foo?

Are we allowed to release foo?

R.20: Use unique_ptr or shared_ptr to represent ownership

Types of smart pointers

unique_ptr

- Very light-weight
- Negligible overhead
- Single owner
- Default smart pointer

shared_ptr

- Uses reference counting
  - Actually, two
- Larger overhead
- Less restrict
- Allows multiple owners
- Should only be used for that
  - But, unfortunately, used too often

shared_ptr instead of unique_ptr

Simpler semantics than unique_ptr

- No need to bother with move

Simple drop in replacement

- Pure stupidity!
- Does not express the **intent**
- Large overhead
  - Space, speed and contention
- Not safer than getting by reference

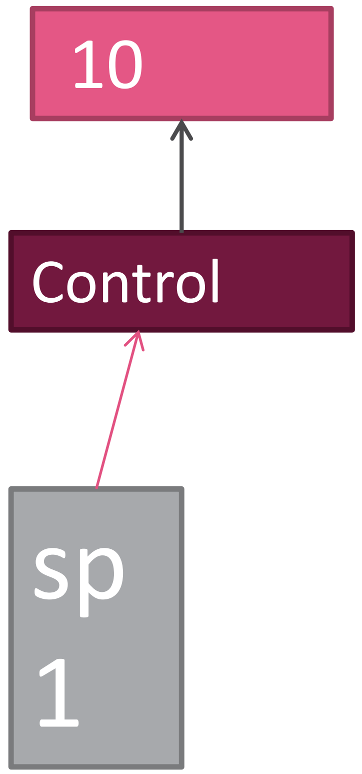
R.21: Prefer unique_ptr over shared_ptr unless you need to share ownership

weak_ptr

- Let me hold reference for the object
- But do not keep the object just for me
  - For example, cache
- Before using, needs to be converted to shared_ptr
  - Using lock() method
- Validity can be checked with expired()

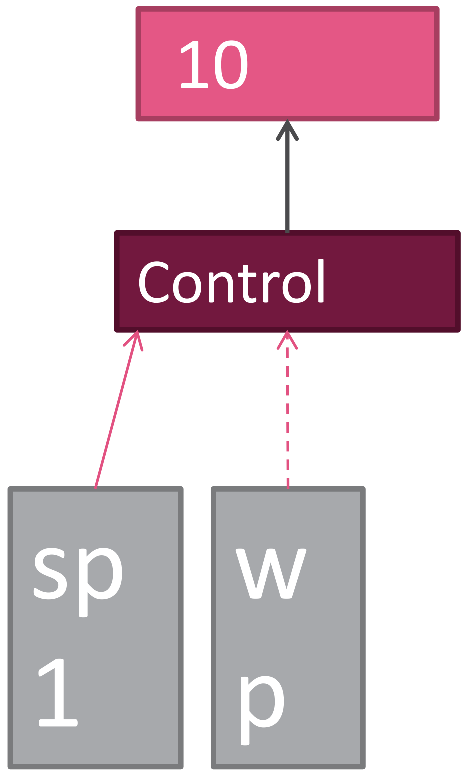
weak_ptr, example

```
auto sp1 = std::make_shared<int>(10);
std::weak_ptr<int> wp (sp);
auto sp2 = wp.lock();
sp2.reset();
sp1.reset();
wp1.reset();
```



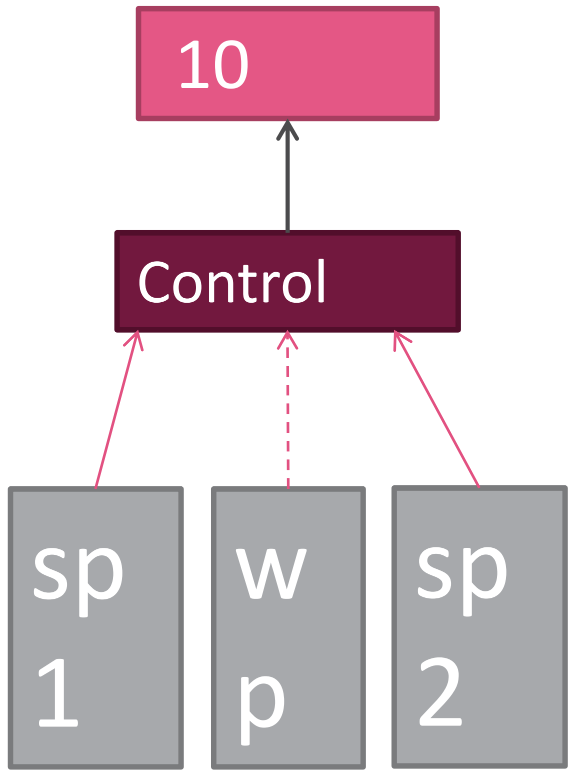
weak_ptr, example

```
auto sp1 = std::make_shared<int>(10);
std::weak_ptr<int> wp (sp);
auto sp2 = wp.lock();
sp2.reset();
sp1.reset();
wp1.reset();
```



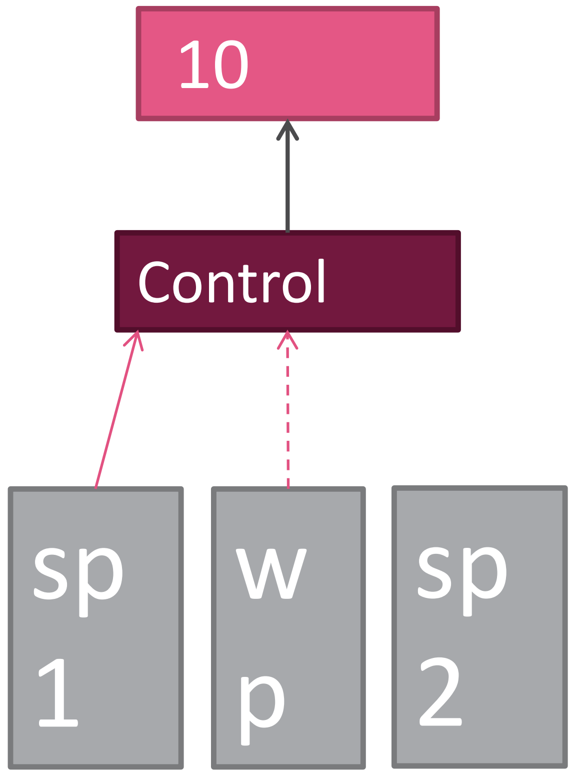
weak_ptr, example

```
auto sp1 = std::make_shared<int>(10);
std::weak_ptr<int> wp (sp);
auto sp2 = wp.lock();
sp2.reset();
sp1.reset();
wp1.reset();
```



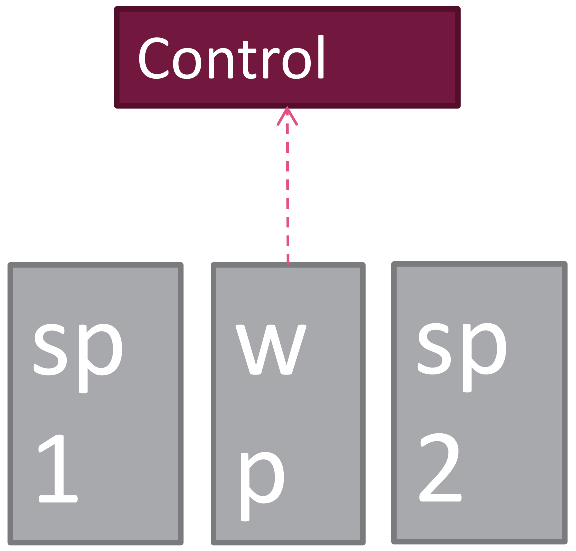
weak_ptr, example

```
auto sp1 = std::make_shared<int>(10);
std::weak_ptr<int> wp (sp);
auto sp2 = wp.lock();
sp2.reset();
sp1.reset();
wp1.reset();
```



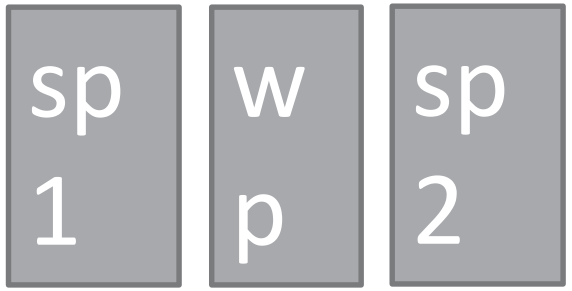
weak_ptr, example

```
auto sp1 = std::make_shared<int>(10);
std::weak_ptr<int> wp (sp);
auto sp2 = wp.lock();
sp2.reset();
sp1.reset();
wp1.reset();
```



weak_ptr, example

```
auto sp1 = std::make_shared<int>(10);
std::weak_ptr<int> wp (sp);
auto sp2 = wp.lock();
sp2.reset();
sp1.reset();
wp1.reset();
```

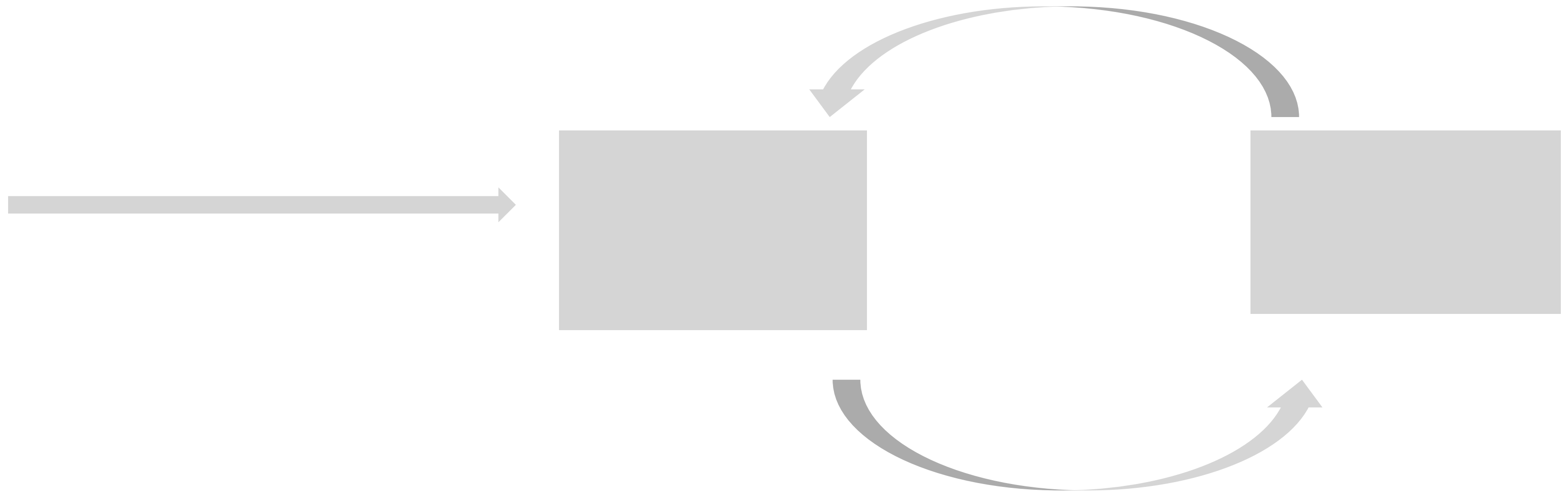


Cyclic reference

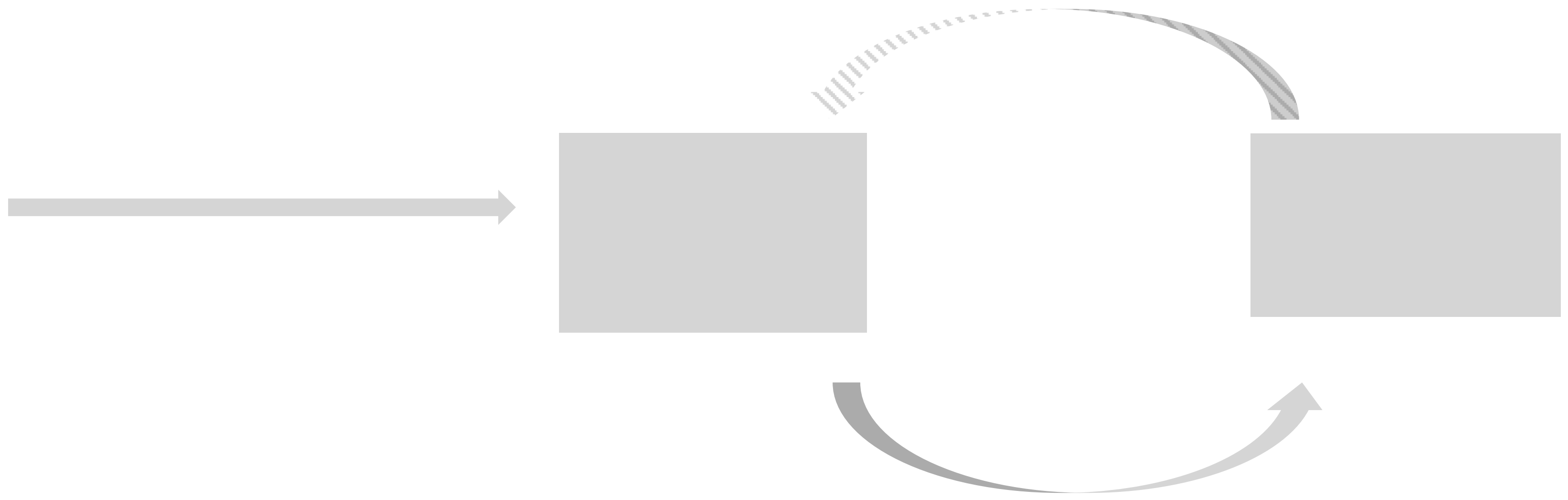
Not very common

Two objects point at each-other

So neither will be released



Resolution: Use weak_ptr



R.24: Use std::weak_ptr to break cycles of shared_ptr s

auto_ptr

Don’t use!

Replace with unique_ptr, if you see it.

Basic usage

unique_ptr, smart_ptr

Creating

```
std::unique_ptr<Foo> foo (new Foo)


auto  bar = std::make_unique<Foo>();
```

```
std::shared_ptr<Foo> foo (new Foo)


auto  bar = std::make_shared<Foo>();
```

make_unique/make_shared

- No new/delete
- No type name repetition
- Exception safer
  - foo (unique_pre<Bar>(new Bar), unique_ptr<baz>(new Baz))
- Faster, one allocation instead of two
  - Less fragmentation, Locality
  - Allocation overhead

but…

- Less flexible
  - No custom deleter
- Cannot be used on existing pointers
  - From legacy code

R.22: Use make_shared() to make shared_ptr s

R.23: Use make_unique() to make unique_ptr s

Redundant temporaries

```
{
    Foo foo;
    auto fooPtr = make_shared<Foo>(foo);

    //do something with fooPtr
    //transfer ownership
   return fooPtr;
}
```

Calling conventions

Transfer ownership, unique_ptr

```
void foo (std::unique_ptr<Bar> bar)
```

- Callee assumes ownership
- Caller loses ownership

```
auto  bar = make_unique<bar>();
foo(std::move(bar));
```

bar must not be used past this point

Share ownership, shared_ptr

```
void foo (std::shared_ptr<Bar> bar)
```

- Callee shares ownership
- Caller maintains ownership

```
auto  bar = std::make_shared<bar>();
foo(bar);
```

bar can still be used here

R.32: Take a unique_ptr<widget> parameter to express that a function assumes ownership of a widget

R.34: Take a shared_ptr<widget> parameter to express that a function is part owner

Pass by reference, unique_ptr

```
void foo (std::unique_ptr<Bar> &  bar)
```

- Does not change ownership
- Indicates that callee may reseat bar

```
auto  bar = std::make_unique<bar>();
foo(bar);
```

Pass by reference, shared_ptr

```
void foo (const std::shared_ptr<Bar> &  bar)
```

- Callee may share ownership
- Indicates that callee may reseat bar

```
auto  bar = std::make_shared<bar>();
foo(bar);
```

R.33: Take a unique_ptr<widget>& parameter to express that a function reseats the widget

R.35: Take a shared_ptr<widget>& parameter to express that a function might reset the shared pointer

View, do not change lifecycle

```
void foo (const std::unique_ptr<Bar> & bar)
```

```
void foo (const std::shared_ptr<Bar> & bar)
```

```
void foo (const Bar & bar)
```

```
void foo (const Bar * bar)
```

Redundant refCount increase: Pass to function that shares ownership

```
void process (std::shared_ptr<Bar>  bar) {
      //Do some work
       enqueueForLaterProcessing(bar);
}
```

Redundant refCount increase: Possible shared ownership

```
void process (std::shared_ptr<Bar> bar) {
    if (needToProcess)
        async_queue.push_back(bar);
}
```

Possible shared ownership

```
void process (const std::shared_ptr<Bar> & bar) {
        if (canProcessFast(bar)
                processFast(bar);
        else
                enqueueForLaterProcessing(bar);
}
```

R.36: Take a const shared_ptr<widget>& parameter to express that it might retain a reference count to the object ???

Other points to discuss

- enable_shared_from_this
- Thread safety
- Reset