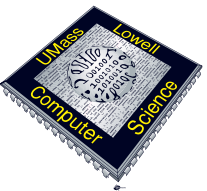
Smart Pointers Part 4

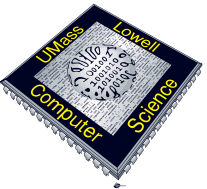
COMP.2040 – Computing IV

Dr. Tom Wilkes



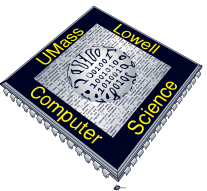
Sources

- Scott Meyers, *Effective Modern C++*, Chapter 4
- Stanley B. Lippman, Josee Lajoie, and Barbara E. Moo, *C++ Primer* (5th edition), Chapter 12
- Stackoverflow <https://stackoverflow.com/questions/12030650/when-is-stdweak-ptr-useful>



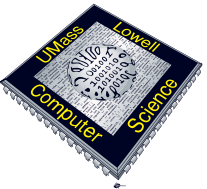
weak_ptrs

- A `weak_ptr` is a smart pointer that does not control the lifetime of the object to which it points.
- Instead, a `weak_ptr` points to an object that is managed by a `shared_ptr`.
- Binding a `weak_ptr` to a `shared_ptr` does not change the reference count of that `shared_ptr`.
- Once the last `shared_ptr` pointing to the object goes away, the object itself will be deleted.
- That object will be deleted even if there are `weak_ptrs` pointing to it—hence the name `weak_ptr`, which captures the idea that a `weak_ptr` shares its object "weakly."



Motivation: Cache Example

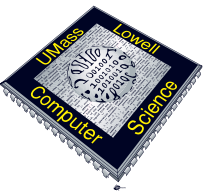
- For recently accessed objects, you want to keep them in memory, so you hold a “strong” pointer to them. Periodically, you scan the cache and decide which objects have not been accessed recently. You don't need to keep those in memory, so you get rid of the strong pointer.
- But what if that object is in use and some other code holds a strong pointer to it? If the cache gets rid of its only pointer to the object, it can never find it again. So the cache keeps a weak pointer to objects that it needs to find if they happen to stay in memory.
- This is exactly what a weak pointer does -- it allows you to locate an object if it's still around, but doesn't keep it around if nothing else needs it.



Source: < https://stackoverflow.com/questions/12030650/when-is-stdweak-ptr-useful>

`weak_ptr` operations

`weak_ptr<T> w`	Null `weak_ptr` that can point at objects of type `T`.
`weak_ptr<T> w(sp)`	`weak_ptr` that points to the same object as the `shared_ptr` `sp`. `T` must be convertible to the type to which `sp` points.
`w = p`	`p` can be a `shared_ptr` or a `weak_ptr`. After the assignment `w` shares ownership with `p`.
`w.reset()`	Makes `w` null.
`w.use_count()`	The number of `shared_ptrs` that share ownership with `w`.
`w.expired()`	Returns `true` if `w.use_count()` is zero, `false` otherwise.
`w.lock()`	If `expired` is `true`, returns a null `shared_ptr`; otherwise returns a `shared_ptr` to the object to which `w` points.

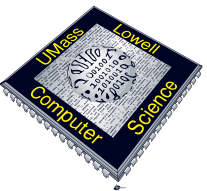


Creating a `weak_ptr`

- When we create a `weak_ptr`, we initialize it from a `shared_ptr`:

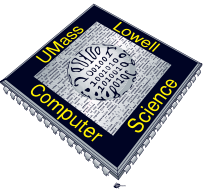
```
auto p = make_shared<int>(42);
weak_ptr<int> wp(p);
// wp weakly shares with p;
// use count in p is unchanged
```

- Here both `wp` and `p` point to the same object. Because the sharing is weak, creating `wp` doesn't change the reference count of `p`; it is possible that the object to which `wp` points might be deleted.



`weak_ptr` and `lock` (1)

- Because the object might no longer exist, we cannot use a `weak_ptr` to access its object directly.
- To access that object, we must call `lock`. The `lock` function checks whether the object to which the `weak_ptr` points still exists. If so, `lock` returns a `shared_ptr` to the shared object.

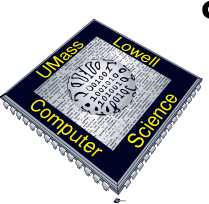


weak_ptr and lock (2)

- As with any other shared_ptr, we are guaranteed that the underlying object to which that shared_ptr points continues to exist at least as long as that shared_ptr exists. For example:

```
if (shared_ptr<int> np = wp.lock())
{ // true if np is not null
  // inside the if, np shares its
  // object with p
}
```

- Here we enter the body of the if only if the call to lock succeeds. Inside the if, it is safe to use np to access that object.

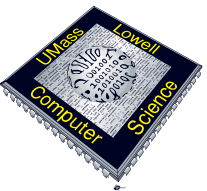


Checked Pointer Example (1)

```
// StrBlobPtr throws an exception on attempts to access
// a nonexistent element
class StrBlobPtr
{
   public:
      StrBlobPtr(): curr(0) {}
      StrBlobPtr(StrBlob &a, size_t sz = 0):
         wptr(a.data), curr(sz) {}
      std::string& deref() const;
      StrBlobPtr& incr(); // prefix version

   private:
      // check returns a shared_ptr to the vector
      // if the check succeeds
      std::shared_ptr<std::vector<std::string>>
        check(std::size_t, const std::string&) const;

      // store a weak_ptr, which means the underlying
      // vector might be destroyed
      std::weak_ptr<std::vector<std::string>> wptr;
      std::size_t curr; // current position within
                        // the array
};
```



Checked Pointer Example (2)

```
std::shared_ptr<std::vector<std::string>>
StrBlobPtr::check(std::size_t i,
                  const std::string &msg) const
{
   auto ret = wptr.lock();
   // is the vector still around?

   if (!ret)
     throw std::runtime_error("unbound
StrBlobPtr");

   if (i >= ret->size())
     throw std::out_of_range(msg);
   return ret;
   // otherwise, return a shared_ptr to
   // the vector
}
```