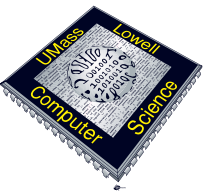
Smart Pointers Part 3

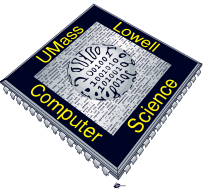
COMP.2040 – Computing IV

Dr. Tom Wilkes



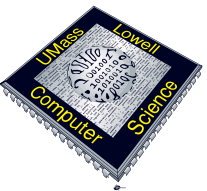
Sources

- Scott Meyers, *Effective Modern C++*, Chapter 4
- Stanley B. Lippman, Josee Lajoie, and Barbara E. Moo, *C++ Primer* (5th edition), Chapter 12



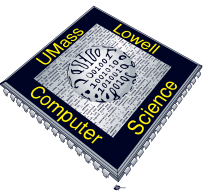
Unique_ptrs

- A `unique_ptr` "owns" the object to which it points.
- Unlike `shared_ptr`, only one `unique_ptr` at a time can point to a given object. The object to which a `unique_ptr` points is destroyed when the `unique_ptr` is destroyed.
- The table on the next slide lists the operations specific to `unique_ptr`s. The operations common to both were covered in a table in the Part 1 slide deck.



`unique_ptr` Operations

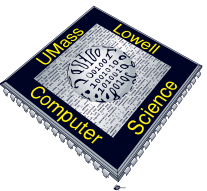
`unique_ptr<T> u1`	Null `unique_ptr`s that can point to objects of type `T`. `u1` will use `delete` to free its pointer; `u2` will use a callable object of type `D` to free its pointer.
`unique_ptr<T, D> u2`	
`unique_ptr<T, D> u(d)`	Null `unique_ptr` that point to objects of type `T` that uses `d`, which must be an object of type `D` in place of `delete`.
`u = nullptr`	Deletes the object to which `u` points; makes `u` null.
`u.release()`	Relinquishes control of the pointer `u` had held; returns the pointer `u` had held and makes `u` null.
`u.reset()`	Deletes the object to which `u` points;
`u.reset(q)`	If the built-in pointer `q` is supplied, makes `u` point to that object.
`u.reset(nullptr)`	Otherwise makes `u` null.



Creating a `unique_ptr`

- Unlike `shared_ptr`, there is no library function comparable to `make_shared` that returns a `unique_ptr`.
- Instead, when we define a `unique_ptr`, we bind it to a pointer returned by `new`.
- As with `shared_ptr`s, we must use the direct form of initialization:

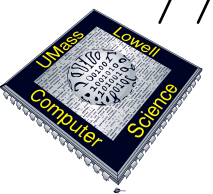
```
unique_ptr<double> p1;
  // unique_ptr that can point at a double

unique_ptr<int> p2(new int(42));
  // p2 points to int with value 42
```



No copy or assign for `unique_ptrs`

- Because a `unique_ptr` owns the object to which it points, `unique_ptr` does not support ordinary copy or assignment:

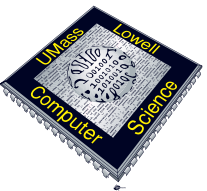
```
unique_ptr<string> p1(new
                      string("Stegosaurus"));

unique_ptr<string> p2(p1);
// error: no copy for unique_ptr

unique_ptr<string> p3;
p3 = p2;
// error: no assign for unique_ptr
```



Release and reset (1)

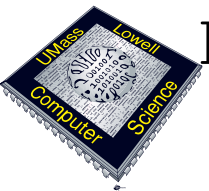
- Although we can't copy or assign a `unique_ptr`, we can transfer ownership from one (non-`const`) `unique_ptr` to another by calling `release` or `reset`:

```
// transfers ownership from p1 (which
// points to the string Stegosaurus) to p2

unique_ptr<string> p2(p1.release());
// release makes p1 null


unique_ptr<string> p3(new string("Trex"));
// transfers ownership from p3 to p2
p2.reset(p3.release());
// reset deletes the memory to which p2
// had pointed
```



`Release` and `reset` (2)

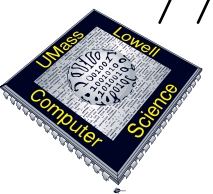
- The `release` member function returns the pointer currently stored in the `unique_ptr` and makes that `unique_ptr` null. Thus, `p2` is initialized from the pointer value that had been stored in `p1` and `p1` becomes null.
- The `reset` member function takes an optional pointer and repositions the `unique_ptr` to point to the given pointer. If the `unique_ptr` is not null, then the object to which the `unique_ptr` had pointed is deleted. The call to `reset` on `p2`, therefore, frees the memory used by the string initialized from "Stegosaurus", transfers `p3`'s pointer to `p2`, and makes `p3` null.



`Release` and `reset` (3)

- Calling `release` breaks the connection between a `unique_ptr` and the object it had been managing.
- Often the pointer returned by `release` is used to initialize or assign another smart pointer. In that case, responsibility for managing the memory is simply transferred from one smart pointer to another.
- However, if we do not use another smart pointer to hold the pointer returned from `release`, our program takes over responsibility for freeing that resource:

```
p2.release();
// WRONG: p2 won't free the memory and
// we've lost the pointer

auto p = p2.release();
// ok, but we must remember to delete(p)
```



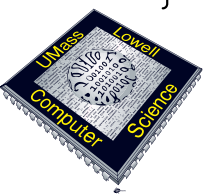
Passing and Returning `unique_ptrs`

- There is one exception to the rule that we cannot copy a `unique_ptr`: We can copy or assign a `unique_ptr` that is about to be destroyed.
- The most common example is when we return a `unique_ptr` from a function:

```
unique_ptr<int> clone(int p)
{
    // ok: explicitly create a unique_ptr<int>
    // from int*
    return unique_ptr<int>(new int(p));
}
```

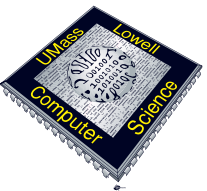
- Alternatively, we can also return a copy of a local object:

```
unique_ptr<int> clone(int p)
{
    unique_ptr<int> ret(new int (p));
    // . . .
    return ret;
}
```



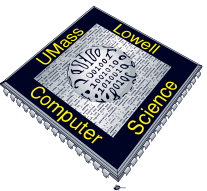
Backward Compatibility: `auto_ptr`

- Earlier versions of the library included a class named `auto_ptr` that had some, but not all, of the properties of `unique_ptr`. In particular, it was not possible to store an `auto_ptr` in a container, nor could we return one from a function.
- Although `auto_ptr` is still part of the standard library, programs should use `unique_ptr` instead.



Passing a Deleter to `unique_ptr` (1)

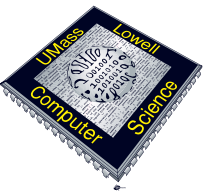
- Like `shared_ptr`, by default, `unique_ptr` uses `delete` to free the object to which a `unique_ptr` points. As with `shared_ptr`, we can override the default deleter in a `unique_ptr`.
- However, the way `unique_ptr` manages its deleter differs from the way `shared_ptr` does. Overriding the deleter in a `unique_ptr` affects the `unique_ptr` type as well as how we construct (or reset) objects of that type.



Passing a Deleter to `unique_ptr` (2)

- Similar to overriding the comparison operation of an associative container, we must supply the deleter type inside the angle brackets along with the type to which the `unique_ptr` can point.
- We supply a callable object of the specified type when we create or reset an object of this type:

```
// p points to an object of type objT and
// uses an object of type delT to free
// that object

// it will call an object named fcn of
// type delT

unique_ptr<objT, delT> p (new objT, fcn);
```



Passing a Deleter to `unique_ptr` (3)

- As a somewhat more concrete example, we'll rewrite our connection program to use a `unique_ptr` in place of a `shared_ptr` as follows:

```
void f(destination &d /* other needed parameters
*/)
{
    connection c = connect(&d);
    // open the connection

    // when p is destroyed, the connection
    // will be closed
    unique_ptr<connection,
               decltype(end_connection)*>
       p(&c, end_connection);

    // use the connection

    // when f exits, even if by an exception,
    // the connection will be properly closed
}
```