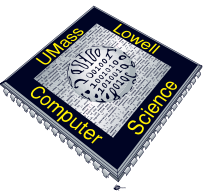
Smart Pointers Part 1

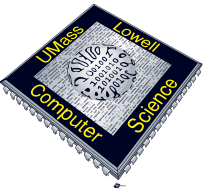
COMP.2040 – Computing IV

Dr. Tom Wilkes



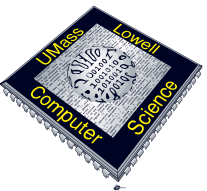
Sources

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



Motivation for Smart Pointers

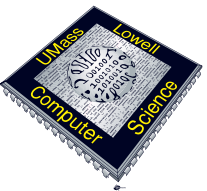
- In C++, dynamic memory is traditionally managed via the pair of operators `new` and `delete`
  - This is problematic because the onus is on the programmer to allocate and free memory at the correct times
- C++11 introduces the concept of **smart pointers** in order to help automate memory management
- Smart pointers are implemented by container classes in the new library (in the `<memory>` header file):
  - `shared_ptr`
  - `unique_ptr`
  - `weak_ptr`



The `shared_ptr` Class

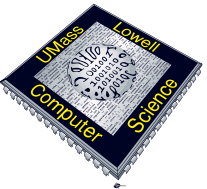
Smart pointers are templates; so, we must supply the type to which the pointer can point.

```
shared_ptr<string> p1;
// shared_ptr that can point at a string

shared_ptr<list<int>> p2;
// shared_ptr that can point at a list of ints


// if p1 is not null, check whether it's the
// empty string

if (p1 && p1->empty())
      *p1 = "hi"; // if so, dereference p1 to
                  // assign a new value to that
                  // string
```



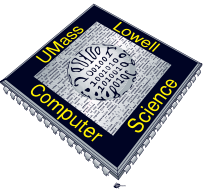
Operations Common to `shared_ptr` and `unique_ptr`

`shared_ptr<T> sp` `unique_ptr<T> up`	Null smart pointer that can point to objects of type `T`.
`p`	Use `p` as a condition; `true` if `p` points to an object.
`*p`	Dereference `p` to get the object to which `p` points.
`p->mem`	Synonym for `(*p).mem`.
`p.get()`	Returns the pointer in `p`. Use with caution; the object to which the returned pointer points will disappear when the smart pointer deletes it.
`swap(p, q)` `p.swap(q)`	Swaps the pointers in `p` and `q`.



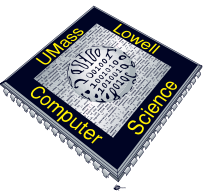
Operations Specific to `shared_ptr`

`make_shared<T>(args)`	Returns a `shared_ptr` pointing to a dynamically allocated object of type `T`. Uses *args* to initialize that object.
`shared_ptr<T> p(q)`	`p` is a copy of the `shared_ptr` `q`; increments the count in `q`. The pointer in `q` must be convertible to `T*` (§ 4.11.2, p. 161).
`p = q`	`p` and `q` are `shared_ptrs` holding pointers that can be converted to one another. Decrements `p`'s reference count and increments `q`'s count; deletes `p`'s existing memory if `p`'s count goes to 0.
`p.unique()`	Returns `true` if `p.use_count()` is one; `false` otherwise.
`p.use_count()`	Returns the number of objects sharing with `p`; may be a slow operation, intended primarily for debugging purposes.



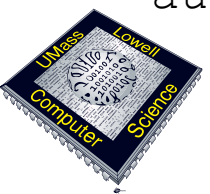
The `make_shared` Function

- The safest way to allocate and use dynamic memory is to call a library function named `make_shared`.
  - This function allocates and initializes an object in dynamic memory and returns a shared_ptr that points to that object.
  - Like the smart pointers, make_shared is defined in the memory header.
- When we call `make_shared`, we must specify the type of object we want to create.
  - We do so in the same way as we use a template class, by following the function name with a type enclosed in angle brackets…



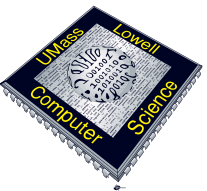
Examples of `make_shared`

```
// p3 is a shared_ptr that points to an
// int with value 42
shared_ptr<int> p3 = make_shared<int>(42);


// p4 points to a string with value
// 9999999999
shared_ptr<string> p4 = make_shared<string>(10, '9');


// p5 points to an int that is value
// initialized to 0
shared_ptr<int> p5 = make_shared<int>();


// p6 points to a dynamically allocated, empty
// vector<string>
auto p6 = make_shared<vector<string>>();
```



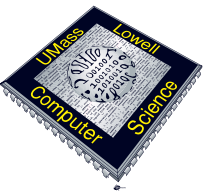
Shared pointer implementation

- We can think of a `shared_ptr` as if it has an associated counter, usually referred to as a reference count.
  - Whenever we copy a `shared_ptr`, the count is incremented.
  - For example, the counter associated with a `shared_ptr` is incremented when we use it to initialize another `shared_ptr`, when we use it as the right-hand operand of an assignment, or when we pass it to or return it from a function by value.
  - The counter is decremented when we assign a new value to the `shared_ptr` and when the `shared_ptr` itself is destroyed, such as when a local `shared_ptr` goes out of scope.



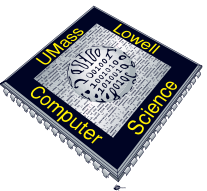
When the shared pointer counter goes to zero…

```
auto r = make_shared<int>(42);
// int to which r points has one user


r = q;
// - assign to r, making it point to a
// different address
// - increase the use count for the
// object to which q points
// - reduce the use count of the
// object to which r had pointed
// - the object r had pointed to has
// no users; that object is
// automatically freed
```



`shared_ptrs` Automatically Destroy Their Objects ...

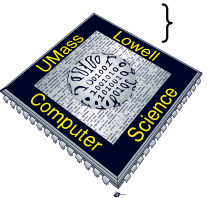
- When the last `shared_ptr` pointing to an object is destroyed, the `shared_ptr` class automatically destroys the object to which that `shared_ptr` points.
  - It does so through another special member function known as a destructor. Analogous to its constructors, each class has a destructor. Just as a constructor controls initialization, the destructor controls what happens when objects of that class type are destroyed.
  - Destructors generally free the resources that an object has allocated. For example, the `string` constructors (and other `string` members) allocate memory to hold the characters that compose the `string`. The `string` destructor frees that memory. Similarly, several `vector` operations allocate memory to hold the elements in the `vector`. The destructor for `vector` destroys those elements and frees the memory used for the elements.
  - The destructor for `shared_ptr` decrements the reference count of the object to which that `shared_ptr` points. If the count goes to zero, the `shared_ptr` destructor destroys the object to which the `shared_ptr` points and frees the memory used by that object.



...and Automatically Free the Associated Memory (1)

- The fact that the `shared_ptr` class automatically frees dynamic objects when they are no longer needed makes it fairly easy to use dynamic memory.
- For example, we might have a function that returns a `shared_ptr` to a dynamically allocated object of a type named `Foo` that can be initialized by an argument of type `T`:

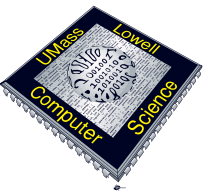
```
// factory returns a shared_ptr pointing
// to a dynamically allocated object

shared_ptr<Foo> factory(T arg)
{
    // process arg as appropriate

    // shared_ptr will take care of
    // deleting this memory
    return make_shared<Foo>(arg);
}
```



...and Automatically Free the Associated Memory (2)

- Because `factory` returns a `shared_ptr`, we can be sure that the object allocated by factory will be freed when appropriate.
- For example, the following function stores the `shared_ptr` returned by `factory` in a local variable:

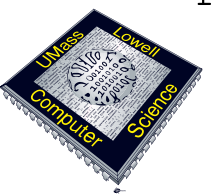
```
void use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // use p
} // p goes out of scope; the memory
to which p points is
  // automatically freed
```



...and Automatically Free the Associated Memory (3)

- Because (in the example on the previous slide) `p` is local to `use_factory`, it is destroyed when `use_factory` ends. When `p` is destroyed, its reference count is decremented and checked. In this case, `p` is the only object referring to the memory returned by `factory`. Because `p` is about to go away, the object to which `p` points will be destroyed and the memory in which that object resides will be freed.
- The memory will not be freed if there is any other `shared_ptr` pointing to it:

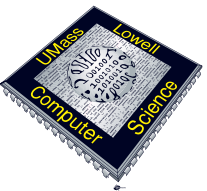
```
shared_ptr<Foo> use_factory(T arg)
{
    shared_ptr<Foo> p = factory(arg);
    // use p
    return p;  // reference count is incremented
when we return p

} // p goes out of scope; the memory to which p
points is not freed
```



Other Ways to Define and Change `shared_ptrs`

`shared_ptr<T> p(q)`	`p` manages the object to which the built-in pointer `q` points; `q` must point to memory allocated by `new` and must be convertible to `T*`.
`shared_ptr<T> p(u)`	`p` assumes ownership from the `unique_ptr` `u`; makes `u` null.
`shared_ptr<T> p(q, d)`	`p` assumes ownership for the object to which the built-in pointer `q` points. `q` must be convertible to `T*` (§ 4.11.2, p. 161). `p` will use the callable object `d` (§ 10.3.2, p. 388) in place of `delete` to free `q`.
`shared_ptr<T> p(p2, d)`	`p` is a copy of the `shared_ptr` `p2` as described in Table 12.2 except that `p` uses the callable object `d` in place of `delete`.

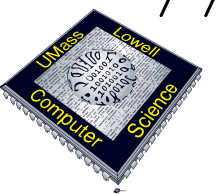
`p.reset()`
`p.reset(q)`
`p.reset(q, d)`
	If `p` is the only `shared_ptr` pointing at its object, `reset` frees `p`'s existing object. If the optional built-in pointer `q` is passed, makes `p` point to `q`, otherwise makes `p` null. If `d` is supplied, will call `d` to free `q` otherwise uses `delete` to free `q`.



Using `shared_ptrs` with `new (1)`

- As we've seen, if we do not initialize a smart pointer, it is initialized as a `null` pointer.
- As described in the table on the previous slide, we can also initialize a smart pointer from a pointer returned by `new`:

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

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

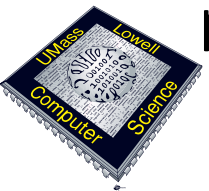


Using `shared_ptrs` with `new (2)`

- The smart pointer constructors that take pointers are `explicit`.
- Hence, we cannot implicitly convert a built-in pointer to a smart pointer; we must use the direct form of initialization to initialize a smart pointer:

```
shared_ptr<int> p1 = new int(1024);
// error: must use direct initialization

shared_ptr<int> p2(new int(1024));
// ok: uses direct initialization
```

- The initialization of p1 implicitly asks the compiler to create a `shared_ptr` from the `int*` returned by `new`.
- Because we can't implicitly convert a pointer to a smart pointer, this initialization is an error.



Using `shared_ptrs` with `new (3)`

- For the same reason, a function that returns a `shared_ptr` cannot implicitly convert a plain pointer in its `return` statement:

```
shared_ptr<int> clone(int p)
{
    // error: implicit conversion to
    // shared_ptr<int>
    return new int(p);
}
```

- We must explicitly bind a `shared_ptr` to the pointer we want to return:

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