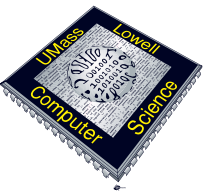
Smart Pointers Part 2

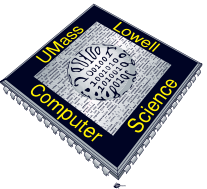
COMP.2040 – Computing IV

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

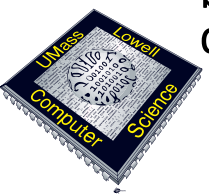


Don’t Mix Ordinary Pointers and Smart Pointers (1)

- A `shared_ptr` can coordinate destruction only with other `shared_ptrs` that are copies of itself.
- There is no way to inadvertently bind the same memory to more than one independently created `shared_ptr`.
- Consider the following function that operates on a `shared_ptr`:

```
// ptr is created and initialized
// when process is called
void process(shared_ptr<int> ptr)
{
   // use ptr

} // ptr goes out of scope and is destroyed
```

- The parameter to `process` is passed by value, so the argument to `process` is copied into `ptr`. Copying a `shared_ptr` increments its reference count. Thus, inside `process` the count is at least 2.
- When `process` completes, the reference count of `ptr` is decremented but cannot go to zero. Therefore, when the local variable `ptr` is destroyed, the memory to which `ptr` points will not be deleted.



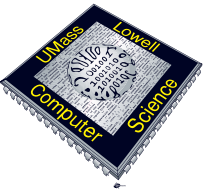
Don’t Mix Ordinary Pointers and Smart Pointers (2)

- The right way to use this function is to pass it a `shared_ptr`:

```
shared_ptr<int> p(new int(42)); // reference count is 1
process(p); // copying p increments its count;
            // in process the reference count is 2
int i = *p; // ok: reference count is 1
```

- Although we cannot pass a built-in pointer to `process`, we can pass process a (temporary) `shared_ptr` that we explicitly construct from a built-in pointer. However, doing so is likely to be an error:

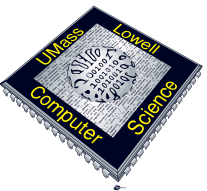
```
int *x(new int(1024)); // dangerous: x is a plain
                       // pointer, not a smart pointer

process(x);  // error: cannot convert int* to
             // shared_ptr<int>

process(shared_ptr<int>(x)); // legal, but the memory
                             // will be deleted!

int j = *x;  // undefined: x is a dangling pointer!
```



Don’t Mix Ordinary Pointers and Smart Pointers (3)

- When we bind a `shared_ptr` to a plain pointer, we give responsibility for that memory to that `shared_ptr`.
- Once we give `shared_ptr` responsibility for a pointer, we should no longer use a built-in pointer to access the memory to which the `shared_ptr` now points.

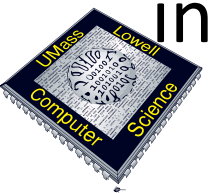
Warning: It is dangerous to use a built-in pointer to access an object owned by a smart pointer, because we may not know when that object is destroyed.



Don't Use `get` to Initialize or Assign Another Smart Pointer (1)

- The smart pointer types define a function named `get` that returns a built-in pointer to the object that the smart pointer is managing.
- This function is intended for cases when we need to pass a built-in pointer to code that can't use a smart pointer.
- The code that uses the return from `get` must not delete that pointer.

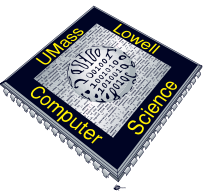
Warning: Use `get` only to pass access to the pointer to code that you know will not delete the pointer. In particular, never use `get` to initialize or assign to another smart pointer.



Don't Use `get` to Initialize or Assign Another Smart Pointer (2)

- Although the compiler will not complain, it is an error to bind another smart pointer to the pointer returned by `get`:

```
shared_ptr<int> p(new int(42)); // reference count is 1
int *q = p.get(); // ok: but don't use q in any way
                  // that might delete its pointer

{ // new block

  // undefined: two independent shared_ptrs point
  // to the same memory
  shared_ptr<int>(q);

} // block ends, q is destroyed, and the memory
  // to which q points is freed

int foo = *p; // undefined; the memory to which p
              // points was freed
```

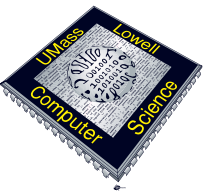


Smart Pointers and Exceptions (1)

- Programs that use exception handling to continue processing after an exception occurs need to ensure that resources are properly freed if an exception occurs.
- One easy way to make sure resources are freed is to use smart pointers.
- When we use a smart pointer, the smart pointer class ensures that memory is freed when it is no longer needed even if the block is exited prematurely:

```
void f()
{
   shared_ptr<int> sp(new int(42)); // allocate a new object

   // code that throws an exception that is not caught inside f

} // shared_ptr freed automatically when the function ends
```

- When a function is exited, whether through normal processing or due to an exception, all the local objects are destroyed.
- In this case, `sp` is a `shared_ptr`, so destroying `sp` checks its reference count.
- Here, `sp` is the only pointer to the memory it manages; that memory will be freed as part of destroying `sp`.

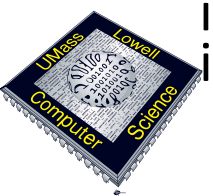


Smart Pointers and Exceptions (2)

- In contrast, memory that we manage directly is not automatically freed when an exception occurs.
- If we use built-in pointers to manage memory and an exception occurs after a new but before the corresponding delete, then that memory won't be freed:

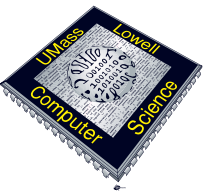
```
void f()
{
   int *ip = new int(42);// dynamically allocate
                         // a new object

    // code that throws an exception that is
    // not caught inside f

   delete ip; // free the memory before exiting
}
```

- If an exception happens between the new and the delete, and is not caught inside `f`, then this memory can never be freed. There is no pointer to this memory outside the function `f`. Thus, there is no way to free this memory.



Smart Pointers and Dumb Classes (1)

- Many C++ classes, including all the library classes, define destructors that take care of cleaning up the resources used by that object.
- However, not all classes are so well behaved. In particular, classes that are designed to be used by both C and C++ generally require the user to specifically free any resources that are used.
- Classes that allocate resources—and that do not define destructors to free those resources—can be subject to the same kind of errors that arise when we use dynamic memory. It is easy to forget to release the resource.
- Similarly, if an exception happens between when the resource is allocated and when it is freed, the program will leak that resource.

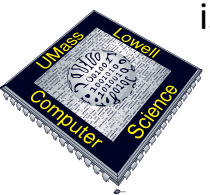


Smart Pointers and Dumb Classes (2)

- We can often use the same kinds of techniques we use to manage dynamic memory to manage classes that do not have well-behaved destructors.
- For example, imagine we're using a network library that is used by both C and C++. Programs that use this library might contain code such as:

```
struct destination; // represents what we are connecting to
struct connection;  // information needed to use the connection
connection connect(destination*);  // open the connection
void disconnect(connection);       // close the given connection
void f(destination &d /* other parameters */)
{
    // get a connection; must remember to close it when done

    connection c = connect(&d);

    // use the connection
    // if we forget to call disconnect before exiting f,
    // there will be no way to close c
}
```

- If `connection` had a destructor, that destructor would automatically close the connection when f completes. However, `connection` does not have a destructor.
- This problem is nearly identical to our previous program that used a `shared_ptr` to avoid memory leaks. It turns out that we can also use a `shared_ptr` to ensure that the connection is properly closed.



Using Our Own Deletion Code

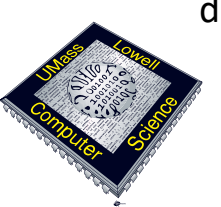
- By default, `shared_ptrs` assume that they point to dynamic memory. Hence, by default, when a `shared_ptr` is destroyed, it executes delete on the pointer it holds.
- To use a `shared_ptr` to manage a connection, we must first define a function to use in place of `delete`. It must be possible to call this **deleter** function with the pointer stored inside the `shared_ptr`. In this case, our **deleter** must take a single argument of type `connection*`:

```
void end_connection(connection *p) { disconnect(*p); }
```

- When we create a `shared_ptr`, we can pass an optional argument that points to a **deleter** function:

```
void f(destination &d /* other parameters */)
{
    connection c = connect(&d);
    shared_ptr<connection> p(&c, end_connection);

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

- When `p` is destroyed, it won't execute `delete` on its stored pointer. Instead, `p` will call `end_connection` on that pointer. In turn, `end_connection` will call `disconnect`, thus ensuring that the connection is closed.
- If `f` exits normally, then `p` will be destroyed as part of the return. Moreover, `p` will also be destroyed, and the connection will be closed, if an exception occurs.



Summary: Smart Pointer Pitfalls

Smart pointers can provide safety and convenience for handling dynamically allocated memory only when they are used properly. To use smart pointers correctly, we must adhere to a set of conventions:

- Don't use the same built-in pointer value to initialize (or `reset`) more than one smart pointer.
- Don't `delete` the pointer returned from `get()`.
- Don't use `get()` to initialize or `reset` another smart pointer.
- If you use a pointer returned by `get()`, remember that the pointer will become invalid when the last corresponding smart pointer goes away.
- If you use a smart pointer to manage a resource other than memory allocated by `new`, remember to pass a **deleter**.