【C++】Day53

Class	C++
Date	@February 7, 2022
Material	
Series Number	
Summary	

【Ch12】Dynamic Memory

12.1.5 unique_ptr

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 following table defines the operations specific to `unique_ptrs`:

Table 12.4. unique_ptr Operations (See Also Table 12.1 (p. 452))

`unique_ptr<T> u1` `unique_ptr<T, D> u2`	Null `unique_ptrs` 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> 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()`
`u.reset(q)`
`u.reset(nullptr)`
	Deletes the object to which `u` points;
	If the built-in pointer `q` is supplied, makes `u` point to that object.
	Otherwise makes `u` null.

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(24)); //o2 points to int with value 42
```

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("Arthur"));
unique_ptr<string> p2(p1); //error: no copy for unique_ptr
unique_ptr<string> p3;
p3 = p1; //error: no assign for unique_ptr
```

Although we cannot copy or assign a `unique_ptr`, we can transfer ownership from one (nonconst) `unique_ptr` to another by calling release or `reset`:

```
unique_ptr<string> p1(new string("Arthur"));
unique_ptr<string> p2(p1.release()); //transfers ownership from p1 to p2
unique_ptr<string> p3(new string("Cici"));
//transfers ownership from p3 to p2
p2.reset(p3.release()); //reset deletes the memory to which p2 had pointed
```

The `release` member 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 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.

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 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));
}
```

Passing a Deleter to unique_ptr

Like `shared_ptr`, by default, `unique_ptr` uses `delete` to free the object to which a `unique_ptr` points.

Overridding the deleter in a `unique_ptr` affects the `unique_ptr` type as well as how we construct(or reset) objects of that 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);
```