【C++】Day51(2)

Class	C++
Date	@February 5, 2022
Material	
Series Number	
Summary	

【Ch12】Dynamic Memory

12.1.3 Using shared_ptrs with new

We can also initialize a smart pointer from a pointer returned by `new`:

```
shared_ptr<int> p1(new int(124)); //p2 points to an int with value 124
```

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
```

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

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> ptr(new int(p));
}
```

By default, a pointer used to initialize a smart pointer must point to dynamic memory because, be default, smart pointers use delete to free the associated object.

Don't Mix Ordinary Pointers and Smart Pointers

A `shared_ptr` can coordinate destruction only with other `shared_ptrs` that are copies of itself. That way, we bind a `shared_ptr` to the object at the same time that we allocate it.

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

```
shared_ptr<int> p(new int(42));
process(p);
```

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(shared_ptr<int>(x)); //legal, but the memory will be deleted.
int j = *x;
```

In this call, we passed a temporary `shared_ptr` to `process`. That temporary is destroyed when the expression in which the call appears finishes. Destroying the temporary decrements the reference count, which goes to zero. The memory to which the temporary points is freed when the temporary is destroyed.

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.

And Don't Use get to Initialize or Assign Another Smart Pointer

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 cannot use a smart pointer. The code that uses the return from get must not delete that pointer.

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

In this case, both `p` and `q` point to the same memory. Because they were created independently from each other, each has a reference count of 1. When the block in which `q` was defined ends, `q` is destroyed. Destroying `q` frees the memory to which `q` points. This makes `p` into a dangling pointer, meaning that what happens when we attempt to use p is undefined.

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

Other shared_ptr Operations

We can use `reset` to assign a new pointer to a `shared_ptr`:

```
p = new int(1024); //error: cannot assign a pointer to a shared_ptr
p.reset(new int(1024)); //ok: p points to a new object
```

Like assignment, `reset` updates the reference counts and, if appropriate, deletes the object to which p points. The reset member is often used together with unique to control changes to the object shared among several `shared_ptrs`. Before changing the underlying object, we check whether we're the only user. If not, we make a new copy before making the change:

```
if(!p.unique())
  p.reset(new int(1024));
*p += newVal; //now that we know we're the only pointer, okay to change this object
```