【C++】Day49(2)

Class	C++
Date	@February 1, 2022
Material	
Series Number	
Summary	

【Ch12】Dynamic Memory

The programs we've written so far have used objects that have well-defined lifetimes.

- Global objects are allocated at program start-up and destroyed when the program ends.
- Local, automatic objects are created and destroyed when the block in which they are defined is entered and exited.
- Local `static` objects are allocated before their first use and are destroyed when the program ends.

In addition to supporting automatic and static objects, C++ lets us allocate objects dynamically. Dynamically allocated objects have a lifetime that is independent of where they are created; they exist until they are explicitly freed.

Warning: Although necessary at times, dynamic memory is notoriously tricky to manage correctly.

12.1 Dynamic Memory and Smart Pointers

In C++, dynamic memory is managed through a pair of operators:

1. `new`, which allocates and optionally initializes, an object in dynamic memory and returns a pointer to that object

2. `delete`, which takes a pointer to a dynamic object, destroys that object, and frees the associated memory.

To make using dynamic memory easier and safer, the new library provides two smart pointer types that manage dynamic objects.

A smart pointer acts like a regular pointer with the important exception that it automatically deletes the object to which it points.

The new library defines two kinds of smart pointers that differ in how they manage their underlying pointers:

1. `shared_ptr`, which allows multiple pointers to refer to the same object
2. `unique_ptr`, which "owns" the object to which it points.

The library also defines a companion class named `weak_ptr` that is a weak reference to an object managed by a `shared_ptr`. All three are defined in the memory header.

12.1.1 The shared_ptr Class

When we create a smart pointer, we must supply additional information-in this case, the type to which the pointer can point.

We supply that type inside angle brackets that follow the name of the kind of smart pointer we are defining:

```
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
```

A default initialized smart pointer holds a null pointer.

We use a smart pointer in ways that are similar to using a ointer:

```
//if p1 is not null, check whether it's the empty string
if(p1 && p1->empty())
  *p1 = "hi";
```

The table below lists operations common to `shared_ptr` and `unique_ptr`:

Table 12.1. 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`.

The following operations are specific to `shared_ptr`:

Table 12.2. 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_ptr`s 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:

```
//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>();
```

Copying and Assigning shared_ptrs

When we copy or assign a `shared_ptr`, each `shared_ptr` keeps track of how many other `shared_ptrs` point to the same object:

```
auto p = make_shared<int>(42); //object to which p points has one user
//object to which p and q points has two users
auto q(p); //p and q point to the same object
```

Once a `shared_ptr`'s counter goes to zero, the `shared_ptr` automatically frees.

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

Note: It is up to the implementation whether to use a counter or another data structure to keep track of how many pointers share state. The key point is that the class keeps track of how many `shared_ptrs` point to the same object and automatically frees that object when appropriate.